Proceedings of the VLDB Endowment

Volume 15, No. 6 – February 2022

PVLDB – Proceedings of the VLDB Endowment

Volume 15, No. 6, February 2022.

All papers published in this issue will be presented at the 48th International Conference on Very Large Data Bases, Sydney, Australia, 2022.

Copyright 2022 VLDB Endowment

Volume 15, Number 6, February 2022

Pages i – vii and 1132 - 1310

ISSN 2150-8097

Available at: http://www.pvldb.org and https://dl.acm.org/journal/pvldb

TABLE OF CONTENTS

Front Matter

Research Papers

PVLDB ORGANIZATION AND REVIEW BOARD - Vol. 15

Review Board

Abolfazl Asudeh (University of Michifan)
Aécio Santos (New York University)
Ahmed Eldawy (University of California, Riverside)
Alexander Hall (RelationalAI)
Alexander J Ratner (University of Washington)
Aline Bessa (New York University)
Alkis Simitsis (Athena Research Center)
Altigran da Silva (Universidade Federal do Amazonas)
AnHai Doan (University of Wisconsin-Madison)
Anna Fariha (Microsoft)
Anton Dignös (Free University of Bozen-Bolzano)
Antonio Cavalcante Araujo Neto (University of Alberta)
Arijit Khan (Nanyang Technological University)
Arvind Arasu (Microsoft)
Babak Salimi (University of California, San Diego)
Bailu Ding (Microsoft Research)
Bertram Ludaescher (University of Illinois)
Bolong Zheng (Huazhong University of Science and Technology)
Brandon Haynes (Gray Systems Lab, Microsoft)
Byron Choi (Hong Kong Baptist University)
Carlo Curino (Microsoft -- GSL)
Carlos Scheidegger (The University of Arizona)
Carsten Binnig (TU Darmstadt)
Ce Zhang (ETH)
Cheng Long (Nanyang Technological University)
Chengfei Liu (Swinburne University of Technology)
Chuan Lei (Instacart)
Chunbin Lin (Amazon AWS)
Curtis Dyreson (Utah State University)
Dan Kifer (Pennsylva State University)
Dana M Van Aken (Carnegie Mellon University)
Daniel Deutch (Tel Aviv University)
Daniel Oliveira (UFF, Brazil)
David Koop (Northern Illinois University)
Davide Mottin (Aarhus University)
Dong Xie (Penn State University)
Eduardo Ogasawara (CEFET-RJ)
Eleni Tzirita Zacharatou (TU Berlin)
Fabio Porto (LNCC)
Faisal Nawab (University of California at Irvine)
Fan Zhang (Guangzhou University)
Fatemeh Nargesian (University of Rochester)
Fei Chiang (McMaster University)
Florin Rusu (UC Merced)
Floris Geerts (University of Antwerp)
Fotis Psallidas (Microsoft)
George Fletcher (Eindhoven University of Technology)
George Papadakis (University of Athens)
Gerhard Weikum (Max-Planck-Institut fur Informatik)
Germain Forestier (University of Haute Alsace)
Guoliang Li (Tsinghua University)
Haipeng Dai (Nanjing University)
Harish Doraiswamy (Microsoft Research India)
Heiko Mueller (DeepReason.ai)
Herodotos Herodotou (Cyprus University of Technology)
Holger Pirk (Imperial College)
Hongzhi Yin (The University of Queensland)
Huiping Cao (New Mexico State University)
Immanuel Trummer (Cornell)
Ioana Manolescu (INRIA and Institut Polytechnique de Paris)
Ippokratis Pandis (Amazon)
Ishtiyaque Ahmad (University of California, Santa Barbara)
Jae-Gil Lee (KAIST)
Jana Giceva (TU Munich)
Jeffrey Xu Yu (Chinese University of Hong Kong)
Jens Teubner (TU Dortmund University)
Jia Zou (Arizona State University)
Jian Pei (Simon Fraser University)
Jianguo Wang (Purdue University)
Jiannan Wang (Simon Fraser University)
Jianxin Li (Deakin University)
Jianye Yang (Central South University)
Jiwon Seo (Hanyang University)
Johannes Gehrke (Microsoft)
Jorge Arnulfo Quiane Ruiz (TU Berlin)
Joseph Near (University of Vermont)
Junhu Wang (Griffith University)
Kaiping Zheng (National University of Singapore)
Kangfei Zhao (The Chinese University of Hong Kong)
Karima Echihabi (Mohammed VI Polytechnic University)
Katja Hose (Aalborg University)
Kenneth A Ross (Columbia University)
Kostas Zoumpatianos (Snowflake Computing)
Lei Zou (Peking University)
Leopoldo Bertossi (Universidad Adolfo Ibanez)
Li Xiong (Emory University)
Lianke Qin (University of California, Santa Barbara)
Lijun Chang (The University of Sydney)
Lin Ma (Carnegie Mellon University)
Long Yuan (Nanjing University of Science and Technology)
Lu Qin (UTS)
Luciano Barbosa (Universidade Federal de Pernambuco)
Marcelo Arenas (Universidad Catolica & IMFD)
Maria Luisa Sapino (U. Torino)
Matteo Lissandrini (Aalborg University)
Matthias Boehm (Graz University of Technology)
Matthias Renz (University of Kiel)
Max Heimel (Snowflake)
Maximilian Schleich (University of Washington)
Meihui Zhang (Beijing Institute of Technology)
Melanie Herschel (Universität Stuttgart)
Michael Abebe (University of Waterloo)
Min Xie (Instacart)
Mirella M Moro (Universidade Federal de Minas Gerais)
Mohamed Sarwat (Arizona State University)
Mohammad Dashti (MongoDB)
Mohammad Javad Amiri (University of Pennsylvania)
Mohammad Sadoghi (University of California, Davis)
Muhammad Aamir Cheema (Monash University)

Nikita Bhutani (Megagon Labs)
Oliver A Kennedy (University at Buffalo, SUNY)
Panos K. Chrysanthis (University of Pittsburgh)
Paolo Missier (Newcastle University)
Parth Nagarkar (NMSU)
Paul Groth (University of Amsterdam)
Peng CHENG (East China Normal University)
Peter Pietzuch (Imperial College London)
Pierangela Samarati (Universita delgi Studi di Milano)
Pinar Karagoz (METU, Turkey)
Pinar Tozun (IT University of Copenhagen)
Prithu Banerjee (UBC)
Raoni Lourenço (New York University)
Raul Castro Fernandez (UChicago)
Ravi Ramamurthy (Microsoft)
Raymond Chi-Wing Wong (Hong Kong University of Science and Technology)
Renata Borovica-Gajic (University of Melbourne)
Reynold Cheng (The University of Hong Kong)
Rui Mao (Shenzhen University)
Ruoming Jin (Kent State University)
Sai Wu (Zhejiang University)
Sainyam Galhotra (University of Chicago)
Sanjay Krishnan (University of Chicago)
Sanjib Kumar Das (Google)
Sayan Ranu (IIT Delhi)
Sebastian Link (University of Auckland)
Semih Salihoglu (University of Waterloo)
Senjuti Basu Roy (New Jersey Institute of Technology)
Sergey Melnik (Google)
Shantanu Sharma (New Jersey Institute of Technology)
Shaoxu Song (Tsinghua University)
Sheng Wang (New York University)
Shimin Chen (Chinese Academy of Sciences)
Shumo Chu (University of California, Santa Barbara)
Shweta Jain (University of Illinois, Urbana-Champaign)
Sibo Wang (The Chinese University of Hong Kong)
Srinivasan Keshav (University of Cambridge)
Steffen Zeuch (DFKI GmbH)
Steven E Whang (KAIST)
Subarna Chatterjee (Harvard University)
Sudip Roy (Google)
Supun C Nakandala (University of California, San Diego)
Tamer Özsu (University of Waterloo)
Tarique A Siddiqui (Microsoft Research)
Thomas Heinis (Imperial College)
Thomas Neumann (TUM)
Tianzheng Wang (Simon Fraser University)
Tien Tuan Anh Dinh (Singapore University of Technology and Design)
Tilmann Rabl (HPI, University of Potsdam)
Ting Yu (Qatar Computing Research Institute)
Torben Bach Pedersen (Aalborg University)
Torsten Grust (Universität Tübingen)
Umar Farooq Minhas (Microsoft Research)
Vasiliki Kalavri (Boston University)
Verena Kantere (National Technical University of Athens)
Victor Zakhary (Oracle)
Vivek Narasayya (Microsoft Research)
Vraj Shah (University of California, San Diego)
Walid G Aref (Purdue)
Wasay Abdul (Harvard)
Wei Wang (Hong Kong University of Science and Technology (Guangzhou))
Wei Lu (Renmin university of china)
Weiren Yu (University of Warwick)
Wen Hua (The University of Queensland)
Wolfgang Lehner (TU Dresden)
Xi He (University of Waterloo)
Xiang Lian (Kent State University)
Xiao Qin (IBM Research)
Xiaofei Zhang (University of Memphis)
Xiaokui Xiao (National University of Singapore)
Xiaolan Wang (Megagon Labs)
Xiaoyang Wang (Zhejiang Gongshang University)
Xin Huang (Hong Kong Baptist University)
Yael Amsterdamer (Bar-Ilan university)
Yanyan Shen (Shanghai Jiao Tong University)
Ye Yuan (Northeastern University)
Yeye He (Microsoft Research)
Yi Chen (NJIT)
Yi Lu (MIT)
Yikai Zhang (Chinese University of Hong Kong)
Yinan Li (Microsoft Research)
Ying Zhang (University of Technology Sydney)
Yongxin Tong (Beihang University)
Yuanyuan Zhu (Wuhan University)
Yue Wang (Shenzhen Institute of Computing Sciences, Shenzhen University)
Yufei Tao (Chinese University of Hong Kong)
Yuliang Li (Megagon Labs)
Yuncheng Wu (National University of Singapore)
Yunjun Gao (Zhejiang University)
Yuval Moskovitch (University of Michigan)
Zhifeng Bao (RMIT University)
Zhongle Xie (Zhejiang University)
Zi Huang (University of Queensland)
Ziawasch Abedjan (Leibniz Universität Hannover)
Zohar Karnin (Amazon)
Zsolt István (IT University of Copenhagen)

LETTER FROM THE EDITORS IN CHIEF

We are pleased to present the sixth issue of PVLDB, Volume 15. This issue contains 14 papers in total including 11 regular research papers and 3 scalable data science (SDS) papers. A broad range of topics are covered in this issue including distributed database systems, machine learning & applied AI for data management, spatial data management, graph data management, database engines, provenance and workflows, data quality, data mining, and information retrieval.

For the first paper in this issue, Wang et al. propose an efficient distributed algorithm for agglomerative hierarchical clustering. Next, Chang et al. present a near-optimal approach for solving the edge connectivity-based hierarchical graph decomposition problem. Tong et al. study the problem of secure spatial queries over data federation and present efficient solutions to solve the problem. Fuchs et al. introduce Sortledton, a universal graph data structure that is optimized for the most relevant data access patterns used by graph computation kernels. Zhang et al. propose NBTree, a lock-free persistent-memory-friendly B+-Tree for eADR-enabled persistent memory systems. Tuli et al. propose TranAD, a deep transformer network-based model for anomaly detection in multivariate time series data. Zhao et al. present the deterministic algorithms to solve the frequency estimation and frequent item problems in the bounded-deletion model. Zhao et al. propose a distributed graph neural network system ByteGNN to support efficient GNN training. Jiang et al. introduce graph neural network models for community search and attributed community search problems. Li et al. present Hyper-Tune, an efficient and robust distributed hyper-parameter tuning framework. Minartz et al. propose efficient algorithms for detecting multivariate correlations in static and streaming data. Poppe et al study the problem of proactive auto-scaling in Microsoft Azure SQL Database Serverless. Cheng et al. propose PGE that leverages both text information and graph structure in product graphs to learn embeddings for error detection. Manne et al. present effective solutions for the multiversion replay problem.

All the papers in this issue will be presented at the 48th International Conference on Very Large Data Bases, 2022, in Sydney. We sincerely thank all the authors for submitting their work and all the reviewers for their outstanding service in reviewing the submissions. We hope that the reader will find this volume enjoyable.

Fatma Özcan, Juliana Freire and Xuemin Lin
Editors-in-Chief of PVLDB Volume 15
Program Chairs for VLDB 2022

PACk: An Efficient Partition-based Distributed Agglomerative Hierarchical Clustering Algorithm for Deduplication

Yue Wang
Microsoft Research
wang.yue@microsoft.com

Vivek Narasayya
Microsoft Research
viveknar@microsoft.com

Yeye He
Microsoft Research
yeyehe@microsoft.com

Surajit Chaudhuri
Microsoft Research
surajitc@microsoft.com

ABSTRACT

The Agglomerative Hierarchical Clustering (AHC) algorithm is widely used in real-world applications. As data volumes continue to grow, efficient scale-out techniques for AHC are becoming increasingly important. In this paper, we propose a Partition-based distributed Agglomerative Hierarchical Clustering (PACk) algorithm using novel distance-based partitioning and distance-aware merging techniques. We have developed an efficient implementation of PACk on Spark . Compared to the state-of-the-art distributed AHC algorithm, PACk achieves 2× to 19× (median=9×) speedup across a variety of synthetic and real-world datasets.

PVLDB Reference Format:
Yue Wang, Vivek Narasayya, Yeye He, and Surajit Chaudhuri. PACk: An Efficient Partition-based Distributed Agglomerative Hierarchical Clustering Algorithm for Deduplication. PVLDB, 15(6): 1132 - 1145, 2022.
doi:10.14778/3514061.3514062

1 INTRODUCTION

Agglomerative Hierarchical Clustering (AHC) is a widely-used clustering algorithm. As noted in the survey article [40], AHC finds applications in different problems including deduplication and record linkage [7, 37, 54, 57], recommender systems [47], bioinformatics [11, 16, 52], computational chemistry [14], environmental science [20], and astronomy [59]. Given an undirected weighted graph $G = (C, W)$, where C is a set of items and W is a set of weighted edges indicating the distances between pairs of items in C, AHC initializes each item into its own cluster and repeatedly merges the next pair of clusters with the smallest distance until no pair of clusters have a distance below a given threshold.

When two clusters are considered for merging in AHC, the distance between the clusters is defined by a *linkage criterion*. A commonly used [40, 62] linkage criterion in practice is the *average* distance over all pairs of edges across items in the two clusters. Other linkage criteria such as minimum (resp. maximum) distance are also used and results in more aggressive (resp. conservative) merging of clusters compared to average distance. Some specialized and efficient algorithms [3, 25, 42, 45, 56] *only* focus on min-linkage which reduces AHC to the simpler minimum spanning tree problem. For our primary motivating scenario of fuzzy deduplication, average-linkage is the most appropriate criterion. Min-linkage is too aggressive and leads to clustering very dissimilar items, and max-linkage tends to be too conservative and results in detection of too few duplicates. Thus, in this paper we focus on the case of average-linkage, which is also a more challenging problem.

Proceedings of the VLDB Endowment, Vol. 15, No. 6 ISSN 2150-8097.
doi:10.14778/3514061.3514062

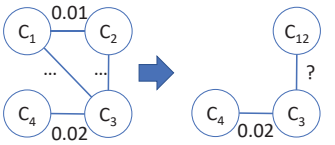


Figure 1: The merge of C_1 and C_2 determines the merge of C_3.

Due to increasing data volumes in real-world applications, the need to support AHC on Big Data platforms such as Spark is growing. For example, the Microsoft Dynamics 365 service applies clustering to find duplicates in their customer profile databases, which can have 100s of millions of records. Centralized (aka single-node) AHC algorithms, which work effectively on relatively small datasets by operating on the data in-memory, are however impractical on large datasets due to their high space complexity ($O(|C|^2)$) and time complexity ($O(|C|^2 \log |C|)$) that lead to memory and CPU bottlenecks on a single machine.

A straightforward adaptation of the centralized AHC algorithms to a scale-out, distributed setting does not perform well because the cost of accessing edges to neighboring nodes in the algorithm becomes excessively high. For example, when the AHC algorithm merges a pair of clusters, it must update distances between the newly merged cluster and all other clusters. In centralized AHC, the update is achieved via a set of relatively cheap writes in memory. However, in a distributed setting with multiple compute nodes (e.g., VMs) working on partitions of the data, this update requires data shuffles across partitions, which is significantly slower. Furthermore, in the distributed setting, multiple iterations of re-partitioning and clustering may be needed thereby amplifying the data shuffle cost.

The natural idea of parallelizing the merge operations holds promise, but is challenging to achieve since merges may have dependencies as illustrated in Figure 1. After the merge of C_1 and C_2, the distance between C_{12} and C_3 determines whether C_3 should be merged with C_{12} or C_4. In other words, the merge of C_3 depends on the merge of C_1 and C_2, thereby introducing difficulty in parallelizing merges when the graph is distributed.

The state-of-the-art distributed AHC algorithm is based on [13]. The authors show that when the linkage criterion satisfies a property called "cluster aggregate inequality" [38], we can concurrently merge all *mutual nearest neighbor pairs* in the graph without affecting correctness of the result. A mutual nearest neighbor pair is a pair of nodes (A, B) such that A is B's nearest neighbor *and* B is A's nearest neighbor. Importantly, they show that this property is

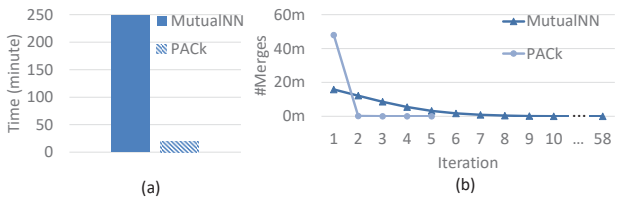


Figure 2: An example real-world dataset with injected duplicates: (a) Our proposed PACk achieves 12× speed-up compared to MutualNN. (b) PACk finishes in 5 iterations, while MutualNN takes 58 iterations with a long tail of fewer merges.

satisfied by linkage criteria including average, max and min – and hence is applicable for our motivating scenarios. An algorithm that exploits this observation, which we refer to as MutualNN is relatively straightforward to implement in a distributed, map-reduce platform using traditional relational operators such as aggregation and join. However, as we show in this paper, this algorithm is inefficient because the number of mutual nearest neighbors is often limited in real-world datasets. Therefore, it often requires multiple (10's) of iterations, with only a few cluster merges possible per iteration as shown in Figure 2. Consequently, MutualNN performs many data scans and shuffles, which lead to large execution time.

In this paper we present the PACk algorithm that builds on the above idea by introducing two novel techniques: distance-based partitioning and distance-aware merging within a partition. Intuitively, the distance-based partitioning algorithm aims to include a set of nearest neighbors in the same partition for each item. It thereby allows more merges to happen within each partition. The distance-aware merging algorithm computes distance bounds to safely merge as many mutual nearest neighbors as possible inside a partition. We show that PACk always produces the same result as centralized AHC. In addition, PACk can perform merges having dependencies in one iteration with guaranteed correctness, which MutualNN cannot do. Our approach parallelizes merges much more effectively and can sharply reduce the number of iterations required, and therefore the overall running time – see Figure 2 for an example on a real-world dataset.

The contributions of this paper are: (1) We present PACk, an efficient distributed clustering algorithm for agglomerative hierarchical clustering. We prove the correctness of PACk, and we have developed an efficient implementation of PACk on Spark. (2) We provide an analytical performance analysis of PACk. We show that PACk is more efficient and needs fewer iterations than MutualNN. (3) We present extensive experimental results comparing the performance and scalability of PACk with MutualNN on a variety of real-world and synthetic graph datasets. PACk consistently outperforms MutualNN with speed-ups ranging from 2× to 19× (median=9×). Its compute resources including CPU and memory are comparable to MutualNN and modestly higher. PACk also scales well to relatively large graphs. For example, on a real-world graph evaluated by the Dynamics 365 service in Microsoft for the task of fuzzy deduplication containing over 250 million items and 680 million edges, PACk finishes in 40 minutes using 16 commodity eight-core VMs, achieving 5× speed-up compared to MutualNN.

We organize the paper as follows: we present the background for AHC in Section 2. We introduce PACk in Section 3. We present the correctness proof and analytical performance analysis in Section 4. We present our experimental evaluation result in Section 5. Finally, we discuss related work in Section 6 and conclude in Section 7.

2 BACKGROUND

2.1 Cluster Labeling and Distance Comparison

When two pairs of clusters have the same distance, we must break ties deterministically to ensure that the result of clustering is deterministic regardless of whether a centralized or distributed AHC algorithm is used. Therefore, in addition to the scalar distance between clusters, we take the cluster labels into comparison.

We assume each initial item $c \in C$ has an associated label denoted by $label(c)$ (e.g., integer, string, etc.). The labels form a totally ordered set. We further assume that the cluster label is the maximum label of the cluster's items:

$$label(C) = \max_{c \in C} \big(label(c)\big) \tag{1}$$

Let $dist(C_i, C_j)$ be the scalar distance between cluster C_i and C_j, so that we can define weight $w(C_i, C_j)$ as a three-element tuple:

Definition 1 (Cluster Pair Edge Weight).

$$w(C_i, C_j) = \big(dist, label1, label2\big)$$

where $dist = dist(C_i, C_j)$, $label1 = \min(label(C_i), label(C_j))$, *and* $label2 = \max(label(C_i), label(C_j))$.

When we compare two edges, we always compare the three-element weight tuples $w(C_i, C_j)$ instead of only the scalar distances $dist(C_i, C_j)$.

Example 1 (Labeling). *Assume we use integer labels in Figure 3 such as* $label(C_1) = 1$, $label(C_2) = 2$, $label(C_{13}) = 3$, *and so on. Two weight examples are* $w(C_1, C_2) = (0.05, 1, 2)$ *and* $w(C_3, C_2) = (0.05, 2, 3)$, *so* $w(C_1, C_2) < w(C_3, C_2)$ *because tuple* $(0.05, 1, 2) < (0.05, 2, 3)$.

2.2 Agglomerative Hierarchical Clustering

Given an undirected weighted graph $G(C, W)$, where C is a set of items and W is a set of weights indicating the distances between pairs of items in C, and a threshold $\theta > 0$, Agglomerative Hierarchical Clustering (AHC) algorithm starts by treating each item as a singleton cluster, iteratively merges nearest cluster pairs, and stops when no two clusters are less than distance θ. Algorithm 1 shows the centralized AHC algorithm, which is straightforward when the entire graph fits in memory.

There exist several linkage criteria to compute the distance function between *clusters* (Line 5) in AHC. Here we list a few:

- Max-linkage (complete-linkage) [12]:
  $dist(C_{ij}, C_x) = \max\big(dist(C_i, C_x), dist(C_j, C_x)\big)$
- Min-linkage (single-linkage) [22, 48]:
  $dist(C_{ij}, C_x) = \min\big(dist(C_i, C_x), dist(C_j, C_x)\big)$
- Average-linkage [50]:
  $dist(C_{ij}, C_x) = \frac{1}{|C_{ij}| \cdot |C_x|} \cdot \sum_{c \in C_{ij}, c' \in C_x} dist(c, c')$

Given its suitability for the fuzzy deduplication problem as noted earlier, in the rest of this paper, we focus on **Average-linkage**.

Algorithm 1: Centralized AHC

Input: Cluster graph $G = (C, W)$; Threshold θ
Output: Clusters C^*

1 **while** *there exists* $dist(C_i, C_j) \leq \theta$ **do**
2 $\quad (C_i, C_j) \Leftarrow \underset{C_i, C_j \in C \wedge C_i \neq C_j}{arg\,min} \left(w(C_i, C_j)\right)$
3 $\quad C_{ij} \Leftarrow C_i \cup C_j$
4 $\quad C \Leftarrow C \cup \{C_{ij}\} - C_i - C_j$
5 $\quad$ Compute $w(C_{ij}, C_x)$ for each $C_x \in C$
6 $C^* \Leftarrow C$

Algorithm 2: `MutualNN`

Input: Cluster graph $G = (C, W)$; Threshold θ
Output: Clusters C^*
`/* Compute in parallel */`

1 **while** *there exists* $dist(C_i, C_j) \leq \theta$ **do**
2 $\quad NN \Leftarrow$ For each C in G, compute its nearest neighbor
3 $\quad MNN \Leftarrow$ Find mutual nearest neighbor pairs by self-join on NN
4 $\quad G \Leftarrow$ Merge mutual nearest neighbors and their edges by join and aggregation

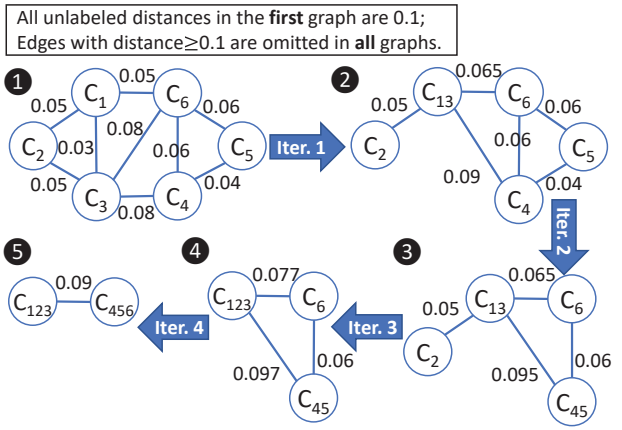


Figure 3: An example that applies Algorithm 1 to six items. θ = 0.08.

Example 2 (Centralized AHC). *Figure 3 is an example that applies Algorithm 1 to six items with threshold $\theta = 0.08$. Edges are labelled with distances. In the* first *graph, unlabeled distances are all 0.1. In* all *the graphs, edges with distance≥0.1 are omitted. The algorithm keeps merging nearest pairs until the remaining edges are greater than* 0.08.

2.3 Distributed AHC

Although the centralized AHC algorithm is straightforward, developing an efficient distributed AHC algorithm is challenging. The efficiency of distributed depends primarily on two factors. The first factor is the number of iterations. Similar to the centralized AHC, a distributed AHC usually takes multiple iterations to finish. In distributed AHC however, each iteration has certain costs such as scanning the graph and writing the intermediate result to persistent storage at the end of the iteration. The second factor is data shuffle. In each iteration, every compute node (VM) works on a partition of a large graph. Since edges can span across partitions, VMs have to shuffle data to find neighbors and update distances. Therefore, techniques that reduce number of iterations and data shuffle cost can lead to greater efficiency and improved performance.

The state-of-the-art distributed AHC is `MutualNN` [13], which parallelizes merges to reduce number of iterations and data shuffle. `MutualNN` is based on the *Cluster Aggregate Inequality* property [13, 38] that makes parallel merges possible. Specifically, in each iteration, `MutualNN` merges all *mutual nearest* neighbor pairs (C_x, C_y) where C_x's nearest neighbor is C_y and vice versa. `MutualNN` guarantees that its result is the same as the centralized AHC as long as the linkage satisfies Cluster Aggregate Inequality:

$$\forall C_i, C_j, C_x : dist(C_{ij}, C_x) \geq \min\left(dist(C_i, C_x), dist(C_j, C_x)\right) \quad (2)$$

The intuition behind Cluster Aggregate Inequality is: if C_x has a unique nearest neighbor C_y, $dist(C_y, C_x)$ must be smaller than any other $dist(C_i, C_x)$ or $dist(C_j, C_x)$, i.e. $dist(C_y, C_x) < \min\left(dist(C_i, C_x), dist(C_j, C_x)\right)$. Hence, merging any other clusters C_i and C_j cannot generate a new cluster C_{ij} whose distance to C_x is closer than $dist(C_y, C_x)$. Therefore, when C_x and C_y are *mutual* nearest neighbors, we can safely merge them. The min-, max-, and average-linkage all satisfy this inequality. The detailed proof of `MutualNN` correctness can be found in [13]. We show why Average-Linkage satisfies the inequality in [55].

Algorithm 2 shows how `MutualNN` works. In each iteration, it finds all mutual nearest neighbor pairs, merges them, and computes the new weights for newly merged clusters. However, `MutualNN` is inefficient as we see in Figure 2, because the number of mutual nearest neighbors is often limited in real-world datasets and `MutualNN` still takes too many iterations. Therefore, we propose a Partition-based distributed Agglomerative hierarchical Clustering (`PACk`) algorithm which significantly increases the number of merges in each iteration to improve the efficiency.

3 PACk ALGORITHM FOR DISTRIBUTED AHC

3.1 Intuition

`PACk` achieves its efficiency using two novel algorithms: distance-based partitioning of the graph, and distance-aware merging within each partition. When partitioning the graph, `PACk` places clusters with their top nearest neighboring clusters together. To limit the size of each partition, for each cluster in a partition, we only include a list of edges with the shortest distances to it, and represent all ignored edges as a lower bound b_L indicating that their distances are greater than b_L. The distance-aware merging algorithm works on each partition and performs merges locally. Whenever it merges a cluster pair, it always ensures that the two clusters are mutual nearest neighbors by checking the distance bounds, which guarantees the correctness of the result.[1] Compared to `MutualNN`, `PACk` performs

[1] Similar to `MutualNN`, `PACk` also works for Max-, Min-, Average-, and any other linkage criterion that satisfies Cluster Aggregate Inequality. The proof of correctness is in Section 4.1.

many more merges in each iteration, thereby reducing the total number of iterations required. Although the shuffle cost for one iteration of PACk could exceed that of MutualNN, since the number of iterations are significantly reduced (Figure 2), the overall shuffle cost of PACk is also much less compared to MutualNN.

Below we provide intuition on why performing more merges in each iteration can improve performance. Observe that for a given input graph, the total number of pair-wise merges done is the same regardless of the specific AHC algorithm used. For instance, given the input in Figure 3, we need four merges to get C_{13}, C_{123}, C_{45}, and C_{456}. Performing more merges in each iteration reduces running time for three reasons: (1) Merges are performed in parallel, which can take less time compared to sequential execution. (2) More merges per iteration reduces the number of iterations, thereby saving the fixed overheads incurred for each iteration. (3) More merges in one iteration reduces the shuffle cost of intermediate results in the following iterations. For example, assume a distributed algorithm finishes in four iterations as shown in Figure 3. It has to generate Graph 2 (resp. Graph 3 and 4) after Iteration 1 (resp. Iter. 2 and 3), and shuffle the graphs' weights to compute nearest neighbor etc. for the next Iteration 2 (resp. Iter. 3 and 4). In comparison, PACk requires one iteration as shown in Figure 4, so we save the shuffle cost of three intermediate graphs, Graph 2, 3, and 4, in Figure 3. Note that the intermediate weights such as $w(C_{13}, C_2)$, $w(C_{123}, C_6)$, etc. are still generated *locally* within each partition, but they are discarded once merging is done for each partition. So these intermediate weights are never shuffled after local merging.

Figure 2b shows an example illustrating how PACk can perform much more merges in one iteration than MutualNN does. It plots the number of merges done by both algorithms on one of our experimental datasets. In the first iteration, PACk completes 99% merges, which is much more than MutualNN's 32%. Moreover, in each of the following iterations, PACk still completes the majority of the *remaining* merges, while MutualNN does only a much smaller percentage. For instance, in the second iteration, PACk does 98% of its remaining merges, while MutualNN does only 37% of its remaining ones. PACk's ability to perform the majority of remaining merges in each iteration significantly reduces the number of iterations and cost of data shuffle, which shortens the running time.

3.2 Overview

Algorithm 3 describes PACk. We assume the input is a graph $G = (C, W)$ where C is the initial item set and W is the weights defined in Section 2.1[2]. PACk keeps merging clusters in iterations as long as the graph has weights that are below the distance threshold. Each iteration consists of four steps:

(1) Partitioning. We partition the graph by putting clusters with their top nearest neighbors together, so that multiple merges have a chance to happen within each partition.

(2) Distance-aware Merging. Within each partition, we merge as many mutual nearest neighbor pairs as possible. For each merge, we track the distance bounds between the newly merged clusters and the other clusters. By comparing the distance bounds, we guarantee to always merge mutual nearest pairs.

(3) Integration. The output clusters of the distance-aware merging need to be integrated because there may be overlapping clusters. As we prove in Theorem 2, every cluster in the output of Algorithm 4 is a correct merge of a mutual nearest pair. Therefore, if two clusters in the output overlap, one must be the superset of the other. Then the integration algorithm keeps the maximal clusters, i.e. those that are not a strict subset of any other cluster.

(4) Graph Update. For each merged cluster, we assign the new label to all its members using a join. Then we aggregate the edges between any two clusters to calculate sum and average distances.

Figure 4 is an example that applies PACk to the same input of Example 2. It finishes in one iteration as we will see in the remainder of Section 3.

Next, we start with the distance-aware merging in Section 3.3, which is a natural extension of the centralized AHC algorithm and allows us to do merge on a partial graph (i.e., a partition). Then, we describe the partitioning algorithm to partition a given graph in Section 3.4.

Algorithm 3: PACk

Input: Cluster graph $G = (C, W)$; Threshold θ
Output: Clusters C^*

```
while there exists dist(C_i, C_j) ≤ θ do
    P ⇐ Partition(G)                          // Algorithm 5 or 6
    C' ⇐ {LocallyMerge(p) | p ∈ P}            // Algorithm 4
    C ⇐ Integrate C'
    W ⇐ Merge weights based on C
C* ⇐ C
```

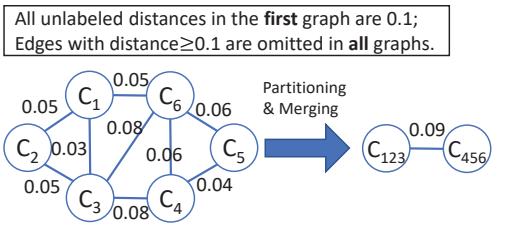


Figure 4: An example that applies PACk to six items. θ = 0.08. PACk finishes in one iteration.

3.3 Distance-Aware Merging

We start with the distance-aware merging algorithm that takes a partition as input and outputs merged clusters. It requires distance bounds as input for each partition, which will be explained in Section 3.4. Developing a merging algorithm that works for a partition and guarantees correctness is challenging because (a) a partition is usually limited by size to fit into a VM's memory, and (b) in one iteration, each partition cannot know the change outside this partition.

We develop a Distance-Aware Merging algorithm that tracks distance bounds to address the above challenges. First, for each cluster C_i in a partition, instead of requiring that all its edges reside in memory, we only require its nearest neighbors to form an edge list $\mathcal{L}(C_i)$ defined below, so that the memory size per partition

[2]In practice, W usually contains only the pairs with meaningful distance (e.g., two strings share at least one token) so that $|W| \ll |C|^2$. Various indexing techniques are used to efficiently retrieve close pairs in different scenarios (e.g., Locality Sensitive Hashing for Jaccard distance, space-partitioning trees for Euclidean distance, n-gram for edit distance, and so on). They are orthogonal to our contribution in this paper.

can be limited. Second, we convert each distance from a scalar to a range, so that the edges outside the partition (i.e. $\notin \mathcal{L}(C_i)$) can be represented by a wildcard edge indicating the lower bound of their distances to C_i. By leveraging the bounds, we are able to safely detect when two clusters are mutually nearest.

Specifically, we define $\mathcal{L}(C_i)$ as the list of nearest neighbors of C_i, whose size limit is a configurable parameter. For each $C_j \in \mathcal{L}(C_i)$, we define $b_L(C_i, C_j)$ and $b_U(C_i, C_j)$ as the lower and upper bounds of $dist(C_i, C_j)$ respectively. $b_L(C_i, C_j)$ and $b_U(C_i, C_j)$ are initialized to $dist(C_i, C_j)$. In addition to the above bounds, we always automatically attach a special wildcard C_i^* into $\mathcal{L}(C_i)$. Its lower bound $b_L(C_i, C_i^*)$ indicates that all remaining neighbors are beyond distance $b_L(C_i, C_i^*)$. Its $b_U(C_i, C_i^*)$ is an application-specific large value (e.g., ∞) indicating the upper bound.

By the definition of Average-linkage in Section 2.2, we can compute the distance between C_{ij} and any other C_x as

$$dist(C_{ij}, C_x) = \frac{dist(C_i, C_x) \cdot |C_i||C_x| + dist(C_j, C_x) \cdot |C_j||C_x|}{(|C_i| + |C_j|) \cdot |C_x|} = \frac{dist(C_i, C_x) \cdot |C_i| + dist(C_j, C_x) \cdot |C_j|}{|C_i| + |C_j|}$$

Similarly, we compute the bounds in three cases:

(1) If a neighbor C_x exists in both $\mathcal{L}(C_i)$ and $\mathcal{L}(C_j)$, we can precisely compute the bounds as:

$$b_L(C_{ij}, C_x) = \frac{b_L(C_i, C_x)|C_i| + b_L(C_j, C_x)|C_j|}{|C_i| + |C_j|}$$
$$b_U(C_{ij}, C_x) = \frac{b_U(C_i, C_x)|C_i| + b_U(C_j, C_x)|C_j|}{|C_i| + |C_j|}$$

(2) If a neighbor C_x exists in only one edge list, say, $\mathcal{L}(C_i)$, we use the wildcard C_j^* for C_j:

$$b_L(C_{ij}, C_x) = \frac{b_L(C_i, C_x)|C_i| + b_L(C_j, C_j^*)|C_j|}{|C_i| + |C_j|}$$
$$b_U(C_{ij}, C_x) = \frac{b_U(C_i, C_x)|C_i| + b_U(C_j, C_j^*)|C_j|}{|C_i| + |C_j|}$$

(3) If a neighbor C_x does not exist in any edge list, we use the wildcard edge (C_{ij}, C_{ij}^*) to represent it. The bounds can be derived from edge (C_i, C_i^*) and (C_j, C_j^*):

$$b_L(C_{ij}, C_{ij}^*) = \frac{b_L(C_i, C_i^*)|C_i| + b_L(C_j, C_j^*)|C_j|}{|C_i| + |C_j|}$$
$$b_U(C_{ij}, C_{ij}^*) = \frac{b_U(C_i, C_i^*)|C_i| + b_U(C_j, C_j^*)|C_j|}{|C_i| + |C_j|}$$

We keep merging clusters within the partition as long as (1) we can find a pair of mutual nearest neighbor (C_i, C_j); and (2) the upper bound of $dist(C_i, C_j)$ (i.e., $b_U(C_i, C_j)$) is no greater than θ. Algorithm 4 shows more detail. It takes a partition P_h and a threshold θ as input. P_h consists of a set of clusters $\mathscr{C}_h$ and the edge lists of the clusters $\{\mathcal{L}(C_i)|C_i \in \mathscr{C}_h\}$. It generates a set of clusters C_{out} as output.

Algorithm 4: Distance-aware Merging for Each Partition

Input: Single partition $P_h = (\mathscr{C}_h, \{\mathcal{L}(C_i)|C_i \in \mathscr{C}_h\})$; Threshold θ
Output: Clusters C_{out}
/* Compute in memory */
1 $G \Leftarrow$ Build a graph from P_h
2 **for** $C_i \in \mathscr{C}_h$ **do**
3 $NN(C_i) \Leftarrow C_i$'s nearest neighbor whose upper bound is smaller than the lower bounds of other C_i's neighbors; *null* if non-existent
4 **while** *true* **do**
5 **if** $\exists (C_i, C_j) : (NN(C_i) = C_j) \wedge (NN(C_j) = C_i) \wedge (b_U(C_i, C_j) \leq \theta)$ **then**
6 Merge C_i with C_j, and update G and $NN(\cdot)$
7 **else**
8 **break**
9 $C_{out} \Leftarrow$ Merged clusters in G

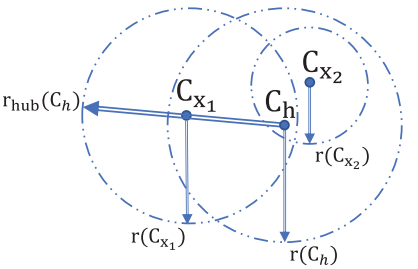


Figure 5: Example of hub radius, in which $r_{hub}(C_h) = r(C_{x_1}) + dist(C_{x_1}, C_h)$.

3.4 Partitioning

In Section 3.3, we show how distance bounds work in distance-aware merging. Next we show how to partition a graph and obtain distance bounds. A partitioning algorithm such as a random partitioning, puts random clusters together in each partition, which likely cannot be merged, thereby rendering it ineffective. Therefore, an effective partitioning algorithm must be carefully designed to let the distance-aware merging perform as many merges as possible for each partition. Intuitively, we want to place clusters with their nearest neighbors together in the same partition, so that more merges can happen locally within that partition. We show in Section 4 that a carefully designed partitioning algorithm needs no more than half the number of iterations of `MutualNN`, which significantly improves the efficiency. We first present the distance-based partitioning Algorithm 5 to illustrate the key idea, and then we present Algorithm 6 that refines Algorithm 5 to allow it to work in practice with the memory constraint of a compute node.

Intuitively, Algorithm 5 puts clusters with their nearest neighbors together to let the merging algorithm perform as many merges as possible. Specifically, Algorithm 5 focuses on clusters that have mutual nearest neighbors (i.e., *hubs*). It creates a partition for each hub by choosing an appropriate *radius* that covers many nearest neighbors but not overlaps other partitions too much.

Algorithm 5 describes the partitioning algorithm. First, it gets a radius for each cluster C_x (Line 11 to 13). The radius is 10 times the distance between C_x and its *second* nearest neighbor (the number 10 helps us reduce number of iterations as we will see in Section 4.3). Now, if we create a partition for each cluster with the radius, the partitions may overlap heavily, and some merges may redundantly

Algorithm 5: Distance-based Partitioning

Input: Cluster graph $G = (C, W)$; Bivariate distance function $dist(\cdot, \cdot)$
Output: Partitions P

```
/* Compute in parallel                                          */
```
1 $Hub \Leftarrow \{C_h | C_h$ has a *mutual* nearest neighbor C'_h and $label(C_h) < label(C'_h)\}$
2 $r_{hub}(\cdot) \Leftarrow$ CalcRadius $(G, Hub, dist(\cdot, \cdot))$ `// Defined below`
3 **foreach** $C_h \in Hub$ **do**
4 $\quad \mathscr{C}_h \Leftarrow \{C_h\} \cup \{C_x | w(C_h, C_x) \in W \ \& \ dist(C_h, C_x) \leq r_{hub}(C_h)\}$
5 $\quad$ **foreach** $C_x \in \mathscr{C}_h$ **do**
6 $\quad\quad \mathcal{L}(C_x) \Leftarrow \{w(C_y, C_x) | dist(C_x, C_y) \leq r_{hub}(C_h)\} \cup \{$wildcard edge for $C_x\}$
7 $\quad P_h \Leftarrow (\mathscr{C}_h, \{\mathcal{L}(C_x) | C_x \in \mathscr{C}_h\})$
`// Each partition` P_h `is a tuple of cluster set and edge list set`
8 $P \Leftarrow \{P_h | C_h \in Hub\}$
9 **return** P

10 **Function** CalcRadius $(G = (C, W), Hub, dist(\cdot, \cdot))$
11 $\quad$ **foreach** $C_x \in C$ **do**
12 $\quad\quad C_y \Leftarrow$ the 2nd nearest neighbor of C_x
13 $\quad\quad r(C_x) \Leftarrow 10 \cdot dist(C_x, C_y)$
14 $\quad$ **foreach** $C_h \in Hub$ **do**
15 $\quad\quad Neighbor(C_h) \Leftarrow \{C_h\} \cup \{C_x | w(C_h, C_x) \in W\}$
16 $\quad\quad r_{hub}(C_h) \Leftarrow \max_{\substack{C_x \in Neighbor(C_h) \\ \& dist(C_h, C_x) \leq r(C_x)}} \left(r(C_x) + dist(C_x, C_h)\right)$

happen in many partitions. So we control the number of partitions by focusing on the "hubs" (Line 1[3]). A hub is a cluster that has a *mutual* nearest neighbor in G, which will be merged because its radius covers its nearest neighbor. In other words, a partition including a hub and its mutual nearest neighbor ensures at least one merge. Therefore, we keep only hubs' partitions but increase the hub radius to cover clusters in other partitions. Specifically, when a hub C_h is covered by another cluster C_x's radius, we increase the hub's radius to cover C_x's radius (Line 14 to 16). Then we can safely ignore all non-hub's partitions. Note that the hubs' partitions may still overlap, but the overlapping space is much smaller and will not hurt efficiency much in practice. Figure 5 shows an example of calculating the hub's radius, in which $r_{hub}(C_h) = r(C_{x_1}) + dist(C_{x_1}, C_h)$ turns out to be the maximum radius for C_h. Finally from Line 3 to 7, for each hub, we collect the clusters within its radius and those clusters' nearest neighbors to form their edge lists.

In theory, Algorithm 5 might create very large partitions that exceeds the memory available on a VM. In order to limit the size of each partition, we use k_N to limit the number of nearest neighbors of hubs, and k_L to limit the size of edge lists. As a result, we simplify Algorithm 5 to get the version with size limit (Algorithm 6). For each hub, it simply gets k_N nearest neighbors and top k_L edges.

Example 3 (Partitioning with Size Limit). *Assume $k_N = k_L = 4$ in Algorithm 6. Given the input graph in Figure 4, we find 2 hubs $\{C_1, C_4\}$.*

C_1 forms a partition P_1 with its nearest 4 neighbors C_3, C_2, C_6, and C_4. When P_1 is passed to Algorithm 4, C_1 and C_3 are firstly merged to get C_{13}. Then C_{13} and C_2 become mutual nearest and are merged into C_{123}.

C_4 forms a partition P_4 with its nearest 4 neighbors C_5, C_6, C_3, and C_1. When P_4 is passed to Algorithm 4, C_1 and C_3 are merged first and $dist(C_{13}, C_6)$ are updated to 0.065. So C_6's nearest neighbor becomes C_4 and is no longer blocked by C_1. Then C_4 and C_5 are merged to C_{45}. Finally C_6 and C_{45} are merged to C_{456}.

In summary, Algorithm 6 creates two partitions P_1 and P_4, which are passed to Algorithm 4 to generate C_{123} and C_{456} respectively. The whole process ends in one iteration.

Algorithm 6: Partitioning with Size Limit

Input: Cluster graph $G = (C, W)$; Bivariate distance function $dist(\cdot, \cdot)$; Neighbor limit k_N; Edge list limit k_L
Output: Partitions P

```
/* Compute in parallel                                          */
```
1 $Hub \Leftarrow \{C_h | C_h$ has a *mutual* nearest neighbor C'_h and $label(C_h) < label(C'_h)\}$
2 **foreach** $C_h \in Hub$ **do**
3 $\quad \mathscr{C}_h \Leftarrow \{C_h\} \cup \{$top k_N neighbors in $W\}$
4 $\quad$ **foreach** $C_x \in \mathscr{C}_h$ **do**
5 $\quad\quad \mathcal{L}(C_x) \Leftarrow \{$top k_L neighbors' weights in $W\} \cup \{$wildcard edge for $C_x\}$
6 $\quad P_h \Leftarrow (\mathscr{C}_h, \{\mathcal{L}(C_x) | C_x \in \mathscr{C}_h\})$
`// Each partition` P_h `is a tuple of cluster set and edge list set`
7 $P \Leftarrow \{P_h | C_h \in Hub\}$

Discussion of k_N and k_L. In practice, moderate k_N and k_L in a wide range like $[50, 500]$ should work reasonably well as we will see in experiments in Section 5.3. If k_N and k_L are too large, it can adversely affect performance because too many distant neighbors are scanned and shuffled without increasing the number of merges in each iteration. Another benefit of using moderate k_N and k_L is to balance the load. For instance, when $k_N = k_L = 500$, the worst-case space of each partition is only around 10 MB ($O(k_N k_L \cdot sizePerEdge)$), and the worst-case time to cluster a partition is only around 10 milliseconds ($O(k_N k_L \cdot \log(k_N k_L))$). In such case, no partition can become a straggler.

4 ANALYSIS OF CORRECTNESS AND PERFORMANCE

In this section we first prove the correctness of PACk and then analyze the performance of the algorithms. Specifically, We propose a Cluster Directed Acyclic Graph (DAG) in Section 4.1 to prove the correctness of PACk. In Section 4.2 and 4.3, we use the DAG to prove that the number of iterations of PACk is half of MutualNN's. In Section 4.4, we use a simplified cost model to show the performance of PACk is better than MutualNN in an example deduplication scenario.

4.1 Cluster DAG and Correctness

Intuitively, we prove the correctness by showing that every cluster we generate must be a merge of two mutual nearest neighbors. It is consistent with the Centralized AHC (Algorithm 1), which also always merges two mutual nearest neighbors.

[3]Hubs can be efficiently detected through a relational group-by query and then a self-join. The group-by scans the graph whose size is $|W|$. Again, $|W| \ll |C|^2$ in practice because users usually remove pairs with long distances. The self-join only joins a table of nearest neighbors of size $|C|$.

We propose a Cluster Directed Acyclic Graph (DAG) that helps us model the number of iterations, which is necessary for estimating the data shuffle cost and running time. **Note** that our algorithm *never* explicitly constructs the DAG during execution. The DAG below is only *conceptual* and for our performance analysis.

Given the set of initial singleton clusters C, the function $dist(\cdot,\cdot)$, and the threshold θ, the DAG $D = (\mathscr{C}, E)$ is constructed based on the execution of Algorithm 1. Specifically, we define *Initial Clusters* as the clusters given in the input graph, define *Merged Clusters* as those merged by Algorithm 1 in all iterations (i.e. C_{ij} in Line 3 of Algorithm 1), and define set $\mathscr{C}$ as the union of all *Initial Clusters* and all *Merged Clusters*. We further define the following functions to help our presentation below: (i) For each Merged cluster C_x, we denote its two direct subclusters by $C^L(C_x)$ and $C^R(C_x)$. Also, we call C_x as the "parent" $C^P(.)$ of its two direct subclusters. E.g., $C^P(C^L(C_x)) = C_x$ and $C^P(C^R(C_x)) = C_x$. (ii) If a cluster is merged with another, we define them as "siblings" $C^S(.)$. For example, $C^S(C^L(C_x)) = C^R(C_x)$ and $C^S(C^R(C_x)) = C^L(C_x)$. For simplicity, we let $C^L(C_x) = C_x^L$, $C^R(C_x) = C_x^R$, $C^P(C_x) = C_x^P$, and $C^S(C_x) = C_x^S$ **hereafter** when the context is clear. Also, we abbreviate nested functions such as $C^L(C^R(C_x)) = C_x^{LR}$ **hereafter**.

The edge set E captures all the dependencies of merges. Specifically, a directed edge (C_x, C_y) means that C_x must be generated before C_y is generated. There are two types of edges: "Subset Dependency" and "Weight Dependency".

Definition 2 (Subset Dependency Edge). *For each Merged cluster C_x, we define two subset dependency edges (C_x^L, C_x) and (C_x^R, C_x).*

Intuitively, Subset Dependency means that C_x^L and C_x^R must be the prerequisites of C_x.

Definition 3 (Weight Dependency Edge). *For each pair of Merged clusters C_x and C_y that satisfies $C_x \cap C_y = \emptyset$, we build an edge (C_x, C_y) if and only if:*

$$\exists C'_y \in \{C_y^L, C_y^R\}, C'_x \in \{C_x^L, C_x^R\} :$$
$$w(C'_y, C'_x) < w(C_y^L, C_y^R)$$

Each Weight Dependency (C_x, C_y) means that C_y cannot be generated yet because C'_y's nearest neighbor is C'_x instead of its sibling $C^S(C'_y)$.

One can view the DAG as one or more binary trees (i.e. dendrograms) plus extra edges: all Initial/Merged clusters and the Subset Dependency edges form one or more binary trees, and the Weight Dependency are the extra edges.

Example 4 (Cluster DAG). *Figure 6 is the Cluster DAG of Example 2. C_1 to C_6 are the Initial singleton clusters. C_{13}, C_{45}, C_{123}, and C_{456} are Merged clusters.*

The solid lines represent Subset Dependency, which in fact form two binary trees (i.e., dendrograms) of the clustering process. For instance, $C^L(C_{13}) = C_1$, $C^R(C_{13}) = C_3$, $C^P(C_{13}) = C_{123}$, and $C^S(C_{13}) = C_2$.

The dashed lines represent Weight Dependency. Weight Dependency (C_{13}, C_{456}) is because C_6's nearest neighbor has been C_1 until the merge (i.e., generation) of C_{13}.

Theorem 1. *The constructed DAG D does not have cycles.*

In order to prove Theorem 1 and to facilitate our following analysis, we define a few concepts.

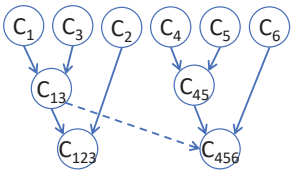


Figure 6: The Cluster DAG of Example 2. Solid lines are subset dependency. Dashed lines are weight dependency.

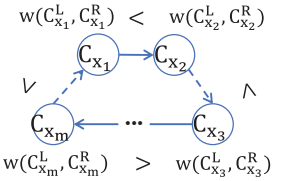


Figure 7: Illustration of the proof of Theorem 1. A cycle will lead to $w(C^L_{x_1}, C^R_{x_1}) < w(C^L_{x_2}, C^R_{x_2}) < ... < w(C^L_{x_m}, C^R_{x_m}) < w(C^L_{x_1}, C^R_{x_1})$, contradiction.

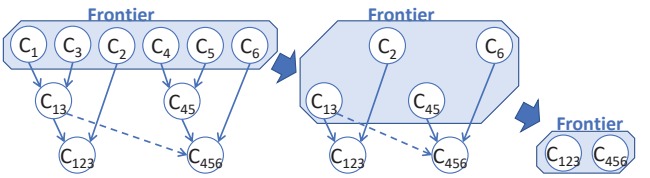


Figure 8: The change of Frontier in Example 5.

Definition 4 (Generated/Ungenerated Clusters). *At the beginning of an algorithm's iteration (i.e. Line 1 of Algorithm 1 or Line 1 of Algorithm 3), a cluster in $\mathscr{C}$ is a Generated Cluster if it is an initial cluster or is already generated through merging by the algorithm. Otherwise, it is an Ungenerated Cluster.*

Definition 5 (Frontier). *At the beginning of an algorithm's iteration, a Frontier F is the set of clusters such that each cluster $C_x \in F$ satisfies* both *conditions below:*

- *C_x is a Generated Cluster.*
- *C_x does not have parent C_x^P, **or** C_x^P is an Ungenerated Cluster.*

In other words, the frontier is the "snapshot" of the clusters at the beginning of each iteration in the algorithm.

Example 5 (Generated/Ungenerated Clusters and Frontier). *Given Example 2, suppose a clustering algorithm finishes in two iterations. The first iteration generates C_{13} and C_{45}. The second iteration generates C_{123} and C_{456}.*

Figure 8 shows how the frontier changes.

Initially, only $\{C_1, C_2, ..., C_6\}$ are Generated, which form the frontier. After the first iteration, C_{13} and C_{45} are Generated. So the frontier becomes $\{C_{13}, C_2, C_{45}, C_6\}$. After the second iteration, all clusters are Generated. The frontier becomes $\{C_{123}, C_{456}\}$.

Now we can prove Theorem 1 by contradiction (Figure 7). The idea is to show that (1) Any cluster C_x in the cycle must be a Merged cluster; (2) Each edge (C_x, C_y) in a cycle satisfy $w(C_x^L, C_x^R) < w(C_y^L, C_y^R)$, leading to a contradiction that $w(C_x^L, C_x^R) < w(C_x^L, C_x^R)$ as we go through the cycle. The detailed proof is in [55].

4.1.1 Correctness. After defining the set of all clusters $\mathscr{C}$, we can prove the correctness of our algorithm.

Theorem 2. *The output of Algorithm 3 is the same as that of Algorithm 1.*

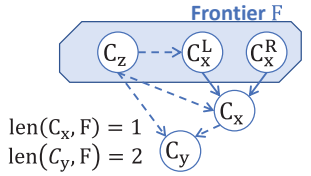


Figure 9: $len(\cdot, F)$ Examples: $len(C_x, F) = 1$ and $len(C_y, F) = 2$.

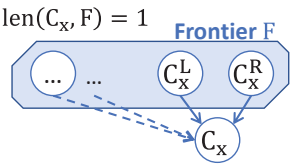


Figure 10: When $len(C_x, F) = 1$, C_x^L and C_x^R must be mutual nearest neighbors in F.

Proof. (Sketch) We prove that (A) every Merged cluster in Algorithm 3 is in the DAG, and that (B) every cluster with 0 out-degree in the DAG is generated by Algorithm 3.

(A) All Merged clusters are generated in Line 6 of Algorithm 4, which guarantees that C_i and C_j are mutual nearest. Therefore, C_i and C_j can be safely merged, because any merge of other clusters won't change the fact that C_i and C_j are mutual nearest neighbors [13, 38]. In addition, when integrating generated clusters (Line 4 in Algorithm 3), we only remove cluster and do not generate extra clusters. So every Merged cluster in Algorithm 3 is in $\mathscr{C}$.

(B) We prove by contradiction. Suppose there exists a cluster with 0 out-degree in the DAG and it is not generated by Algorithm 3. We check all its direct dependents. There are two cases:

(1) All its dependents are generated.
(2) At least one dependent is ungenerated. Let it be C_x. Then we check the dependents of C_x.

We keep checking the dependents of the ungenerated cluster until all dependents are generated. This process should always stop because all Initial clusters are Generated by definition.

When we find the ungenerated C_x whose dependents are all generated, its direct children C_x^L and C_x^R must be mutual nearest in the graph because C_x's dependents are all generated (by Definition 3). Then the partitioning algorithm should build a partition for C_x^L and C_x^R, and Line 6 of Algorithm 4 should generate $C_x = C_x^L \cup C_x^R$. **Contradiction**.

So every cluster with 0 out-degree in the DAG is generated by Algorithm 3. □

4.2 Number of Iterations of MutualNN

We define length to facilitate our proofs below. Note that length is defined only on the DAG, which is irrelevant to the distance function. We define $len(C_x)$ as the longest distance from any *Initial* cluster to $C_x \in \mathscr{C}$, and $len_{max} = \max_{C_x \in \mathscr{C}}(len(C_x))$.

Example 6 (Length from Initial Cluster). *In the DAG in Figure 6, $len(C_1) = len(C_2) = ... = len(C_6) = 0$, $len(C_{13}) = len(C_{45}) = 1$, and $len(C_{123}) = len(C_{456}) = 2$. So $len_{max} = 2$.*

Given a frontier F and an Ungenerated cluster C_x, we define $len(C_x, F)$ as the distance between F to C_x. Formally, let $pa = (C_z, ..., C_x)$ be a *valid* path from C_z to C_x where *only* the first cluster C_z is in F (i.e. $F \cap pa = \{C_z\}$). Let PA be the set of such paths. $len(C_x, F) = \max_{pa \in PA}(\text{length of } pa)$, i.e., the maximum length of these paths.

Example 7. *Figure 9 is an example where Frontier $F = \{C_z, C_x^L, C_x^R\}$.*

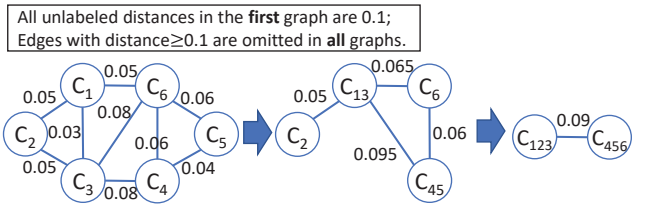


Figure 11: An example that applies MutualNN to six items. $\theta = 0.08$. MutualNN finishes in two iterations.

Regarding C_x, there are three valid paths in PA: (C_z, C_x), (C_x^L, C_x), and (C_x^R, C_x). So $len(C_x, F) = 1$. Note that (C_z, C_x^L, C_x) is not a valid path in PA because the second cluster C_x^L is also in F, violating the definition.

Regarding C_y, there are four valid paths in PA: (C_z, C_y), (C_z, C_x, C_y), (C_x^L, C_x, C_y), and (C_x^R, C_x, C_y). So $len(C_y, F) = 2$.

Theorem 3. *Given a dataset, the number of iterations of* MutualNN *is len_{max} of the DAG.*

The idea is to prove that each iteration of MutualNN generates all clusters C_x with $len(C_x, F) = 1$. Before that, we need to prove a lemma:

Lemma 1. *Given a Merged cluster C_x whose direct children are C_x^L and C_x^R, if there exists a C_y such that $w(C_y, C_x^L) < w(C_x^L, C_x^R)$ OR $w(C_y, C_x^R) < w(C_x^L, C_x^R)$, the DAG must have a path including two edges (C_y, C_y^P) and (C_y^P, C_x).*

The lemma above means, if C_y is closer to any of C_x's children than the child's sibling, there must be a path from C_y to its parent C_y^P to C_x. The detailed proof is in [55].

Now we can prove Theorem 3.

Proof. In each iteration, given a C_x that satisfies $len(C_x, F) = 1$, C_x^L and C_x^R must be mutual nearest in F. Otherwise, if there exists a $C_y \in F$ that is closer to C_x^L (or C_x^R), there should be a path (C^y, C_y^P, C_x) according to Lemma 1. Since $C_y \in F$, C_y^P must be Ungenerated, resulting in $len(C_x, F) \geq 2$, violating $len(C_x, F) = 1$, contradiction. So C_x will be generated when $len(C_x, F) = 1$.

So after iteration t, MutualNN generates all the clusters with $len(C_x) \leq t$. Therefore, MutualNN generates the whole DAG after len_{max} iterations, which is the length of the longest paths. □

Example 8 (MutualNN). *Figure 11 is an example that applies* MutualNN *to six items. It takes two iterations, which equals the length of the longest paths in Cluster DAG in Figure 6.*

4.3 Number of Iterations of PACk

In this section, we assume metric space in which the distance function satisfies triangle inequality, then we can prove that our algorithm takes much fewer iterations than MutualNN. But note that metric space is *not* a requirement of the correctness of PACk.

First we prove that, if the $dist(\cdot, \cdot)$ between *individual* items satisfy triangle inequality, the $dist(\cdot, \cdot)$ between *clusters* also satisfy triangle inequality in Average-Linkage.

Theorem 4. $dist(C_i, C_k) \leq dist(C_i, C_j) + dist(C_j, C_k)$

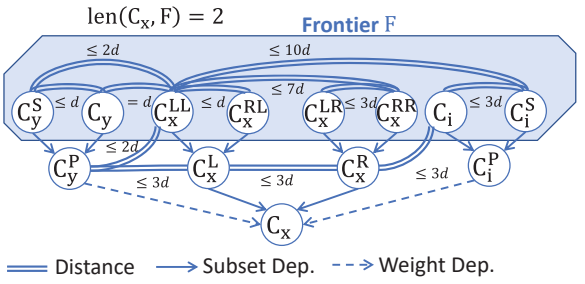


Figure 12: When $len(C_x, F) = 2$, C_x's children are not in F, and C_x^{LL}'s nearest neighbor in F is C_y, Algorithm 3 will generate C_x. (Double lines indicate distances.)

We prove it by enumerating the distances (detail in [55]).

Theorem 5. *Algorithm 3 finishes in $\lceil len_{max}/2 \rceil$ iterations when it uses the Distance-based Partitioning (Algorithm 5) and $dist(\cdot,\cdot)$ is a metric.*

Similar to the proof for `MutualNN`, we can prove that each iteration in Algorithm 3 generates all clusters C_x with $len(C_x, F) \leq 2$.

Lemma 2. *In each iteration, Algorithm 3 generates all clusters C_x that satisfy $len(C_x, F) \leq 2$ where F is the frontier.*

Now we can prove Lemma 2 as the following proof sketch. More detail is in [55].

Proof. (Sketch) When $len(C_x, F) = 1$ (Figure 10), similar to the proof of `MutualNN`, C_x^L and C_x^R must be mutual nearest neighbors in F. Then C_x will be generated in the iteration.

When $len(C_x, F) = 2$, depending on whether C_x's children are in F, there are three cases. In each case, we will prove that C_x's dependents will be in C_x's partition so that C_x's children will become mutual nearest and be merged. The detailed proof is in [55].

Here we briefly present one situation in Case (3), which shows why we set the radius to 10 times the distance between C_x^{LL} and its second nearest neighbor. In Case (3), when C_x^{LL}'s second nearest neighbor is not C_x^R's child. Let C_x^{LL}'s second nearest neighbor be C_y, and $dist(C_x^{LL}, C_y) = d$. Then we can bound many distances as in Figure 12. The longest distance is $dist(C_x^{LL}, C_i^S) \leq 10d$, where C_i^S is a cluster whose parent C_i^P has a weight dependency (C_i^P, C_x). □

Using Lemma 2, we now prove Theorem 5.

Proof. (Sketch) We can prove by induction that, in i-th iteration, Algorithm 3 generates clusters C_x with $len(C_x) \leq 2i$. So in the $\lceil len_{max}/2 \rceil$-th iteration, all clusters are generated. □

We now derive the following corollary for Algorithm 6.

Corollary 1. *If $dist(\cdot,\cdot)$ is a metric, and k_N and k_L are large enough that every $|\mathscr{C}_h|$ and $|\mathcal{L}(C_x)|$ in Algorithm 6 are no less than the corresponding $|\mathscr{C}_h|$ and $|\mathcal{L}(C_x)|$ in Algorithm 5 respectively, Algorithm 3 using Partitioning with Size Limit (Algorithm 6) finishes in $\lceil len_{max}/2 \rceil$ iterations.*

4.4 Simplified Cost Model

In this section, we develop a simplified cost model for the running time τ in the distributed system, and then compare the cost of `MutualNN` and `PACk`.

We define τ as the sum of running time of all iterations:

$$\tau = \sum_{i=1}^{\#iterations} (\tau_s(G_i) + \tau_c(G_i))$$

In Iteration i, where G_i is the input graph, the running time consists of two main parts: data shuffle time $\tau_s(G_i)$, and cpu time $\tau_c(G_i)$. Following the cost models for distributed systems like Spark [2] and Hadoop [23, 24] that assume almost uniform distribution of data[4], we further define:

$$\tau_s(G_i) = \frac{size(G_i)}{networkSpeed \cdot \#executors}$$

where $size(G_i)$ is the size of the graph in bytes, $networkSpeed$ is the number of bytes the system can shuffle in every unit time, and $\#executors$ is the number of executors; and

$$\tau_c(G_i) = \frac{oper(G_i) \cdot timePerOperation}{\#executors}$$

where $oper(G_i)$ is the number of cpu operations to process the graph and $timePerOperation$ is the time for each cpu operation. In practice, $\#executors$ is much less than the number of partitions.

Example Deduplication Scenario. Now we present the cost in a simplified scenario. Assume a set of original items S, each of which has du duplicates. In practice, each item is very similar to its duplicates (e.g., distance ≤ 0.05), but its duplicates are less similar to each other (e.g., distance > 0.05). This is a "hub-spoke" graph where each original item is a hub and its duplicates are spokes. Each item and its duplicates can be viewed as a *group*. To simplify the analysis, further assume that groups are very different from one another, which means no edge across groups as their distances are above the threshold. Let the input graph be G_1 with initial singleton clusters $C(G_1)$ and edge weights $W(G_1)$. So $|C(G_1)| = |S| \cdot (du+1)$, and $|W(G_1)| = |S| \cdot \frac{(du+1)du}{2}$.

Next, we use the Big Theta (Θ) notation to represent the asymptotic complexity.

`MutualNN`. In each iteration, all duplicates find their nearest neighbor (NN) as the original item, but the original item's NN is only one duplicate. So `MutualNN` merges only one pair within each group. Therefore, for the i-th iteration, $C(G_i) = |S| \cdot (du+2-i)$, and $|W(G_i)| = |S| \cdot \frac{(du+2-i)(du+1-i)}{2}$. It takes du iterations to finish.

Each iteration has 3 steps: (1) Find NN through an aggregation. (2) Find mutual NN through a join of the NN pairs. (3) Merge mutual NN and their edges. More detailed cost is in [55], and we only present the total costs here due to space limit:

$$size(G_i) = \Theta((du+2-i)^2|S|) \cdot sizePerEdge$$
$$oper(G_i) = \Theta((du+2-i)^2|S|)$$

[4] We empirically evaluate the performance on skewed data in Section 5.1.

Therefore,

$$\tau = \sum_{i=1}^{\#iterations} (\tau_s(G_i) + \tau_c(G_i))$$

$$= \Theta(du^3)|S| \cdot \left(\frac{sizePerEdge}{networkSpeed \cdot \#executors} + \frac{timePerOperation}{\#executors} \right)$$

Intuitively, the τ contains $\Theta(du^3)|S|$ because the graph has $\Theta(du^2)|S|$ edges initially and has $\Theta((du-i)^2)|S|$ edges after the i-th iteration. The algorithm stops after du iterations, resulting in $\sum_{i=1}^{du} \Theta((du-i)^2)|S| = \Theta(du^3)|S|$ complexity.

PACk. In practice, since *identical* items (like "Seattle" vs "Seattle") are usually aggregated together before clustering happens, so the number of different duplicates (like "Seattle" vs misspelled "Seattt**t**le") is usually very small (e.g., $du \leq 100$). Thus, k_N and k_L are very likely $\geq du$, meaning each item and its duplicates end up in the same partition. So the algorithm finishes in only 1 iteration.

The only iteration has 3 major steps on G_1 (due to space limit, the detailed minor steps are in [55]): partitioning, distance-aware merging, and cluster integration with graph update. The total cost of all steps is:

$$size(G_1) = \Theta(du^2 \cdot |S|) \cdot sizePerEdge$$

$$oper(G_1) = \Theta(|S| \cdot (du^2 \log du))$$

Therefore,

$$\tau = \tau_s(G_1) + \tau_c(G_1)$$

$$= \Theta(du^2)|S| \cdot \frac{sizePerEdge}{networkSpeed \cdot \#executors} + \Theta(du^2 \log du)|S| \cdot \frac{timePerOperation}{\#executors}$$

Comparison. PACk is less expensive than MutualNN in both data shuffle and CPU time in this example. In data shuffle, which is usually orders of magnitude slower than CPU computation, PACk's $\Theta(du^2)$ term in complexity saves much more time than MutualNN's $\Theta(du^3)$. In CPU computation, PACk's $\Theta(du^2 \log du)$ is also better than MutualNN's $\Theta(du^3)$. The $\Theta(\log du)$ term is usually very small in practice (e.g., ≤ 10), because it is bounded by k_N and k_L. So PACk is more efficient in this example.

In other real-world graphs, the graph structure is more complex than this example scenario. There could be random edges with long distances across different groups, which are difficult to be captured by the cost model. So we present experimental evaluations on the real-world data to compare the performance. As we will see in the evaluation, PACk still notably out-performs the state-of-the-art MutualNN on real-world datasets.

5 EVALUATION

In this section we evaluate PACk's (1) performance, (2) scalability and (3) sensitivity to key parameters of the algorithm and compare it with the state-of-the-art algorithm MutualNN.

Dataset. We evaluated PACk (Section 3) on six real, five modified-real and one synthetic datasets shown in Table 1.

The six real datasets are Song, Cite, LiveJ, Wiki, Urban, and Bright. Song and Cite use Jaccard distance and are from the Magellan Data Repository [8]. Song has the titles, releases, and artist names of 1.0M songs. We tokenize each string into a set of tokens, and then keep the pairs of sets with Jaccard distances ≤ 0.4 to get 1.1M edges. Cite is the union of Citeseer and DBLP paper titles containing 4.3M items. Similar to Song, we also tokenize and keep the pairs with Jaccard distance ≤ 0.4 to get 1.9M edges. LiveJ and Wiki are large graphs from Stanford Large Network Dataset Collection [29]. We assign random distances following uniform distribution in $(0, 1]$ for the edges. LiveJ [1, 30] is an online social network with 4.8M items and 69.0M edges. Wiki [27, 58] is the hyperlink network between articles in the most popular categories. It has 1.8M items and 28.5M edges. Urban and Bright use Euclidean distance. Urban is a dataset of road accidents within Great Britain urban areas from the UCI Machine Learning Repository [15]. We keep pairs within 0.5 km to get a graph with 0.4M items and 6.5M edges. Bright [5] is a location-based social network dataset from [29]. We retrieve the distinct locations and keep pairs within 0.5 km to get a graph with 0.8M items and 24.0M edges.

Table 1: Datasets. Numbers in parentheses are of skewed data.

Data	Type	#Items	#Edges
Song	Real	1.0M	1.1M
Cite		4.3M	1.9M
LiveJ		4.8M	69.0M
Wiki		1.8M	28.5M
Urban		0.4M	6.5M
Bright		0.8M	24.0M
Real1	Modified-Real	258.4M (35.4M)	680.4M (20.9M)
Real2		76.8M (10.5M)	358.3M (34.9M)
Real3		19.7M (2.7M)	107.7M (3.2M)
USPS		99.5M (13.6M)	427.4M (39.4M)
IMDB		11.4M (1.6M)	41.0M (1.2M)
FEBRL	Synthetic	10.0M (10.0M)	124.7M (11.6M)

We additionally use six datasets to freely vary the number of duplicates, making the task more challenging. They include five modified-real datasets (Real1, Real2, Real3, USPS, and IMDB) and one synthetic dataset (FEBRL). The Real1, Real2, and Real3 are proprietary datasets used by three different applications in Microsoft. They include names, addresses and other contact information of organizations. USPS is a dataset of addresses in the United States, from which we extract distinct concatenation of street address, city, state and zip code. IMDB contains movie data from the Internet Movie Data Base, in particular the Title, Directors and Genres columns. FEBRL [6] is a synthetic dataset generated using an open source tool. We extract person name, address, suburb, state, and postcode columns from it.

We generate duplicates for these six datasets following uniform and skewed distributions. In the uniform setting, we create 9 similar items for each original item by inserting or deleting random characters. Then we perform a self-join on the data and keep the pairs with Jaccard distance ≤ 0.4. As shown in Table 1, the input graphs has 10.0 to 258.4 million items, with 41.0 to 680.4 million edges. In the skewed setting, we make the number of duplicates follow the Zipfian distribution where the exponent= 3. Then we again keep the pairs with Jaccard distance ≤ 0.4. As shown in the parentheses in Table 1, the input graphs has 1.6 to 35.4 millions items, with 1.2 to 39.4 million edges. (We also evaluate the performance when Jaccard distance ≤ 0.2 in the uniform setting in [55].)

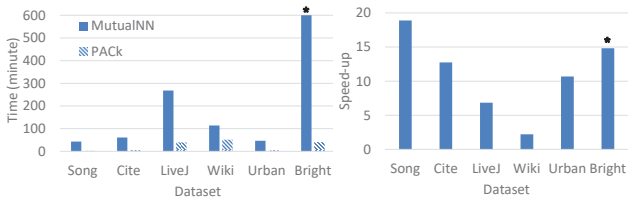


Figure 13: PACk is much more efficient than MutualNN on real datasets; The speed-up ranges from 2.2× to 18.9×. (*: MutualNN exceeds 10 hours on Bright, meaning the speed-up on Bright ≥ 14.8×.)

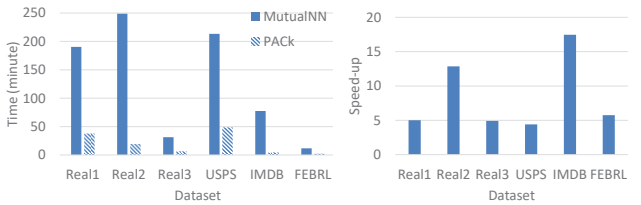


Figure 14: PACk is much more efficient than MutualNN on modified-real and synthetic datasets when number of duplicates follows uniform distribution; The speed-up ranges from 4.4× to 17.4×.

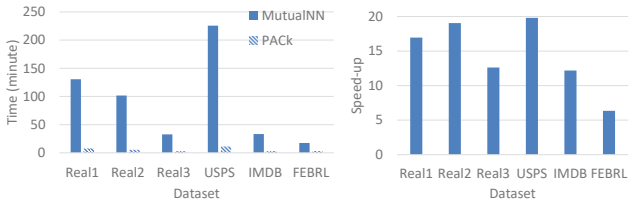


Figure 15: PACk is much more efficient than MutualNN on modified-real and synthetic datasets when number of duplicates follows Zipfian distribution; The speed-up ranges from 6.4× to 19.8×.

Baseline. We compare with the state-of-the-art algorithm that merges mutual nearest neighbors (MutualNN in Section 2.3) in each iteration. The idea was proposed in [38] and later simplified and implemented in [13].

Setting. We conduct experiments on Azure Databricks Spark clusters. The cluster has 16 D8s_v3 virtual machines. Each VM has 8 cores and 32 GB memory, running Apache Spark 2.4.3 and Scala 2.11. The default values of k_N and k_L are 500.

5.1 Performance

Our algorithm is more efficient than the state-of-the-art MutualNN on various datasets. Specifically, we see 2.2× to 18.9× speed-up on the six real datasets (Figure 13), 4.4× to 17.4× speed-up on the six modified-real and synthetic datasets in uniform setting (Figure 14), 6.4× to 19.8× speed-up on the modified-real and synthetic datasets in Zipfian setting (Figure 15).

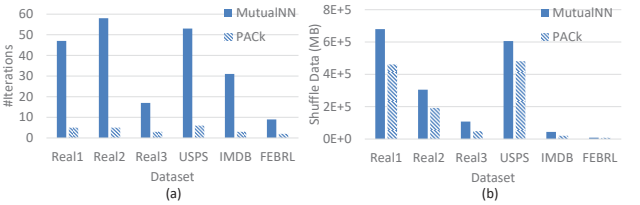


Figure 16: PACk is more efficient than MutualNN because (a) PACk takes fewer iterations; (b) PACk shuffles less data.

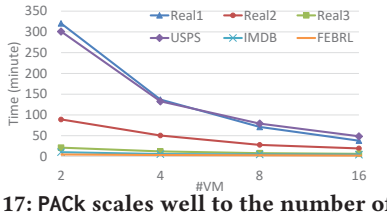


Figure 17: PACk scales well to the number of VMs.

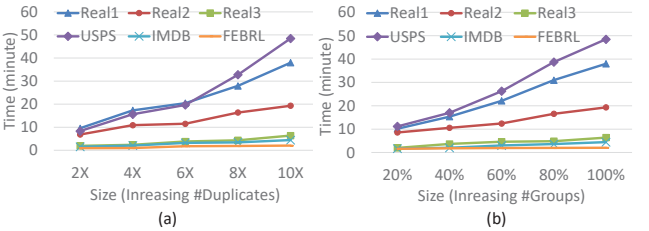


Figure 18: PACk scales almost linearly with the size of data in terms of (a) #duplicates per group; (b) #groups.

PACk is more efficient than MutualNN because PACk finishes in fewer iterations and shuffles less data. For example, on the six modified-real and synthetic datasets in uniform setting, PACk takes only 8.6% to 22.2% of iterations in MutualNN (Figure 16a), and PACk shuffles 45.2% to 88.2% of the data in MutualNN (Figure 16b).

5.2 Scalability

In this experiment we evaluate the scalability of PACk. We vary the number of VMs from 2 to 16. As Figure 17 illustrates, PACk scales well when we vary the number of VMs. For example, on Real1 dataset, the running time using 4 VMs is roughly half the time using 2 VMs. As we increase the VMs, the curve of running time flattens. It is because the merges within each iteration are almost exhaustively parallelized, while the data shuffle between iterations gradually dominates the running time.

Next, we vary the number of items per duplicate group from 2 to 10 (i.e. duplicates from 1 to 9). As Figure 18a shows, PACk scale almost linearly with the number of duplicates.

Next, we vary the number of duplicate groups from 20% to 100% of the original input (fixing the number of items per group at 10). PACk scale almost linearly with the number of groups (Figure 18b).

5.3 Parameter Sensitivity

In this experiment we study how parameters in PACk impact performance. Recall that, in Algorithm 6, k_N controls the size of each

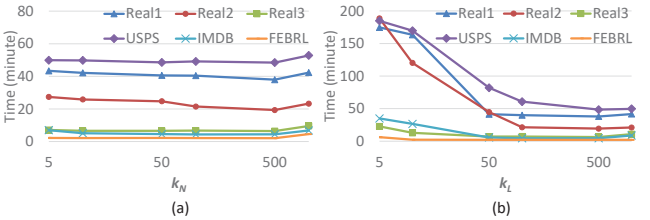


Figure 19: (a) When $k_L = 500$, time slightly decreases as k_N grows because more neighbors are included in a partition; but increases for overly large $k_N = 1000$. (b) When $k_N = 500$, time decreases as k_L grows because more edges are included in a partition; but increases for overly large $k_L = 1000$.

partition, and k_L controls the size of each edge lists. We begin by varying the parameters to see how running time changes.

We first fix $k_L = 500$ and vary k_N in $\{5, 10, 50, 100, 500, 1000\}$. As Figure 19a shows, the running time decreases slightly as k_N grows, because more neighbors are included in a partition and more merges can be done. The time then increases slightly for overly large $k_N = 1000$ as too many neighbors only adds the shuffle cost.

We then fix $k_N = 500$ and vary k_L in $\{5, 10, 50, 100, 500, 1000\}$. As Figure 19b shows, the running time decreases as k_L grows, because more edges are included in a partition and more merges can be done. The time then increases slightly for overly large $k_L = 1000$ because too many edges only adds the shuffle cost.

6 RELATED WORK

The study of Agglomerative Hierarchical Clustering (AHC) dates back to 1950s with a focus on centralized algorithms [21, 26, 28, 39, 40, 49]. The idea is to initially treat each node as a singleton cluster, and then iteratively merge small clusters into bigger clusters based on their pair-wise distances. When two clusters are merged, their distance to a third cluster is updated according to their individual distances. There exist several strategies. For example, Single-linkage [22, 48] takes the minimum distance; Complete-linkage [12] takes the maximum distance; Average-linkage [50] uses the unweighted or weighted average distance. Others strategies include Minimax [4]. These papers assume a centralized AHC that stores the graph in memory. They do not scale to large datasets since they are limited by the compute resources (CPU and memory) available on a single machine.

Researchers have also developed distributed AHC algorithms. In 2005, Ding and He [13] proposed multi-level hierarchical clustering (`MutualNN`), which merges mutually nearest cluster pairs concurrently. They proved that `MutualNN` generates the same result as centralized AHC as long as the distance function satisfies Cluster Aggregate Inequality, which is a stronger version of "reducibility property" [38] proposed in 1980s. Sun et al. [51] implemented AHC using Map-Reduce. Their algorithm collects top K edges with maximal weights from worker nodes to the driver node and merges as many pairs as possible in the centralized driver node. Its scalability is limited to the data size that can fit in the driver node. As a comparison, `PACk` performs clustering distributedly in worker nodes, which has better scalability. Zhang et al. [60, 61] solved AHC under Euclidean distance. They utilize a quad-tree or kd-tree to partition vertices in the Euclidean space, cluster them within each partition, and finally merge clusters. Their technique cannot be generalized to other distance functions like cosine, Jaccard, etc.

Some *approximate* algorithms reduce the running time by sacrificing accuracy. Ma et al. [35] merge multiple clusters in each iteration as long as their distances are within a predefined threshold. A few papers [9, 10, 31] merge edges whose distances are less than an increasing threshold in iterations. Tanaseichuk et al. [52] applies K-means to group items into clusters first and then uses AHC within each cluster. Gilpin et al. [17] group items in Euclidean space into buckets and then apply AHC within each bucket. These approximate algorithms produce different clustering results than conventional AHC does.

Another line of work focuses on special cases of AHC or special computational settings. Single-linkage as a special case of AHC is similar to the typical minimum spanning tree problem. Several efficient distributed single-linkage algorithms have been proposed [3, 25, 42, 45, 56]. Dash et al. [10] proposed an algorithm using shared memory architecture. Some researchers developed a parallel AHC on shared-memory [43] or SIMD machines [32, 33, 46].

Other partitioning strategies exist in some graph systems. For example, Pregel [36] performs message passing between vertices to perform computation on a graph. Each vertex and its neighbors can be viewed as a trivial partition, and its message passing can be supported as data shuffle on Spark using GraphX [19]. In addition, Distributed GraphLab [34] performs edge-cut and PowerGraph [18] performs vertex-cut to partition graphs. These strategies do not leverage the domain knowledge for AHC such as mutual nearest neighbors and distance bounds. In comparison, `PACk` is particularly designed for AHC and has better performance both analytically and in practice.

Many academic and industrial tools support centralized AHC. The examples include MATLAB, R [41], ScikitLearn [44] and SciPy [53] in Python, etc. Similar to the centralized AHC in the papers above, the scalability of these tools is limited to the size of data that can fit in a single node.

7 CONCLUSION

We propose an efficient, distributed agglomerative hierarchical clustering (AHC) algorithm `PACk` that scales well to large data sets. `PACk` derives its efficiency from novel distance-based partitioning and distance-aware merging techniques that enable significantly more merges to be performed in parallel, thereby reducing the number of iterations required as well as the data shuffle cost. We implement `PACk` on Spark, and compare it to the state-of-the-art approach. Our evaluation on several synthetic and real-world datasets including Microsoft Dynamics 365 shows that `PACk` achieves consistently large speedups ranging from 2× to 19× with a median of 9×.

ACKNOWLEDGMENTS

We thank Silu Huang, Wentao Wu, Chi Wang, and Arnd Christian König for their insightful comments on the paper, and Swapna Akula, Katchaguy Areekijseree, Meiyalagan Balasubramanian, and Lengning Liu for their design, implementation, and optimization in Microsoft Dynamics 365 Customer Insights.

REFERENCES

[1] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. Group formation in large social networks: Membership, growth, and evolution. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, page 44–54, New York, NY, USA, 2006. Association for Computing Machinery.

[2] L. Baldacci and M. Golfarelli. A cost model for spark sql. *IEEE Transactions on Knowledge and Data Engineering*, 31(5):819–832, 2019.

[3] M. Bateni, S. Behnezhad, M. Derakhshan, M. Hajiaghayi, R. Kiveris, S. Lattanzi, and V. Mirrokni. Affinity clustering: Hierarchical clustering at scale. In *Advances in Neural Information Processing Systems*, pages 6864–6874, 2017.

[4] J. Bien and R. Tibshirani. Hierarchical clustering with prototypes via minimax linkage. *Journal of the American Statistical Association*, 106(495):1075–1084, 2011.

[5] E. Cho, S. A. Myers, and J. Leskovec. Friendship and mobility: user movement in location-based social networks. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1082–1090, 2011.

[6] P. Christen. Febrl - an open source data cleaning, deduplication and record linkage system with a graphical user interface. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, page 1065–1068, 2008.

[7] W. W. Cohen and J. Richman. Learning to match and cluster large high-dimensional data sets for data integration. In *Proc. SIGKDD*, page 475–480, 2002.

[8] S. Das, A. Doan, P. S. G. C., C. Gokhale, P. Konda, Y. Govind, and D. Paulsen. The magellan data repository. https://sites.google.com/site/anhaidgroup/useful-stuff/data.

[9] M. Dash, H. Liu, P. Scheuermann, and K. L. Tan. Fast hierarchical clustering and its validation. *Data Knowl. Eng.*, 44(1):109–138, Jan. 2003.

[10] M. Dash, S. Petrutiu, and P. Scheuermann. Ppop: Fast yet accurate parallel hierarchical clustering using partitioning. *Data Knowl. Eng.*, 61(3):563–578, June 2007.

[11] Daxin Jiang, Chun Tang, and Aidong Zhang. Cluster analysis for gene expression data: a survey. *IEEE Transactions on Knowledge and Data Engineering*, 16(11):1370–1386, 2004.

[12] D. Defays. An efficient algorithm for a complete link method. *The Computer Journal*, 20(4):364–366, 1977.

[13] C. Ding and X. He. Cluster aggregate inequality and multi-level hierarchical clustering. In *Knowledge Discovery in Databases: PKDD*, pages 71–83, 2005.

[14] G. M. Downs and J. M. Barnard. Clustering methods and their uses in computational chemistry. *Reviews in computational chemistry*, 18:1–40, 2002.

[15] D. Dua and C. Graff. UCI machine learning repository, 2017.

[16] M. B. Eisen, P. T. Spellman, P. O. Brown, and D. Botstein. Cluster analysis and display of genome-wide expression patterns. *Proceedings of the National Academy of Sciences*, 95(25):14863–14868, 1998.

[17] S. Gilpin, B. Qian, and I. Davidson. Efficient hierarchical clustering of large high dimensional datasets. In *Proc. CIKM*, page 1371–1380, 2013.

[18] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 17–30, 2012.

[19] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 599–613, 2014.

[20] P. Govender and V. Sivakumar. Application of k-means and hierarchical clustering techniques for analysis of air pollution: A review (1980–2019). *Atmospheric Pollution Research*, 11(1):40–56, 2020.

[21] J. C. Gower. A comparison of some methods of cluster analysis. *Biometrics*, 23(4):623–637, 1967.

[22] J. C. Gower and G. J. Ross. Minimum spanning trees and single linkage cluster analysis. *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, 18(1):54–64, 1969.

[23] H. Herodotou. Hadoop performance models. Technical report, http://www.cs.duke.edu/starfish/files/hadoop-models.pdf, 2011.

[24] H. Herodotou and S. Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *Proc. VLDB Endow.*, 4(11):1111–1122, Aug. 2011.

[25] C. Jin, R. Liu, Z. Chen, W. Hendrix, A. Agrawal, and A. Choudhary. A scalable hierarchical clustering algorithm using spark. In *Proc. BIGDATASERVICE*, page 418–426, 2015.

[26] S. C. Johnson. Hierarchical clustering schemes. *Psychometrika*, 32(3):241–254, 1967.

[27] C. Klymko, D. F. Gleich, and T. G. Kolda. Using triangles to improve community detection in directed networks. In *The Second ASE International Conference on Big Data Science and Computing, BigDataScience*, 2014.

[28] G. N. Lance and W. T. Williams. A General Theory of Classificatory Sorting Strategies: 1. Hierarchical Systems. *The Computer Journal*, 9(4):373–380, 1967.

[29] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[30] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. *Internet Mathematics*, 6(1):29 – 123, 2009.

[31] K. Li, Y. He, and K. Ganjam. Discovering enterprise concepts using spreadsheet tables. In *SIGKDD*, page 1873–1882, 2017.

[32] X. Li. Hierarchical clustering on simd machines with alignment network. In *Proceedings CVPR '89: IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 660–665, 1989.

[33] X. Li. Parallel algorithms for hierarchical clustering and cluster validity. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(11):1088–1092, 1990.

[34] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, Apr. 2012.

[35] X.-L. Ma, H.-F. Hu, S.-F. Li, H.-M. Xiao, Q. Luo, D.-Q. Yang, and S.-W. Tang. Dhc: Distributed, hierarchical clustering in sensor networks. *Journal of Computer Science and Technology*, 26:643–662, 07 2011.

[36] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146, 2010.

[37] A.-A. Mamun, T. Mi, R. Aseltine, and S. Rajasekaran. Efficient sequential and parallel algorithms for record linkage. *Journal of the American Medical Informatics Association*, 21(2):252–262, 2014.

[38] F. Murtagh. Complexities of hierarchic clustering algorithms: state of the art. *Computational Statistics Quarterly*, 1(2):101–113, 1984.

[39] F. Murtagh and P. Contreras. Algorithms for hierarchical clustering: an overview. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 2(1):86–97, 2012.

[40] F. Murtagh and P. Contreras. Algorithms for hierarchical clustering: an overview, ii. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 7(6):e1219, 2017.

[41] D. Müllner. fastcluster: Fast hierarchical, agglomerative clustering routines for r and python. *Journal of Statistical Software, Articles*, 53(9):1–18, 2013.

[42] V. Olman, F. Mao, H. Wu, and Y. Xu. Parallel clustering algorithm for large data sets with applications in bioinformatics. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 6(2):344–352, 2009.

[43] C. F. Olson. Parallel algorithms for hierarchical clustering. *Parallel Computing*, 21(8):1313 – 1325, 1995.

[44] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[45] S. Rajasekaran. Efficient parallel hierarchical clustering algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 16(6):497–502, 2005.

[46] E. M. Rasmussen and P. Willett. Efficiency of hierarchic agglomerative clustering using the icl distributed array processor. *Journal of Documentation*, 45:1–24, 1989.

[47] A. Shepitsen, J. Gemmell, B. Mobasher, and R. Burke. Personalized recommendation in social tagging systems using hierarchical clustering. In *Proceedings of the 2008 ACM Conference on Recommender Systems*, RecSys '08, page 259–266, New York, NY, USA, 2008. Association for Computing Machinery.

[48] R. Sibson. SLINK: An optimally efficient algorithm for the single-link cluster method. *The Computer Journal*, 16(1):30–34, 1973.

[49] P. H. Sneath. The application of computers to taxonomy. *Microbiology*, 17(1):201–226, 1957.

[50] R. Sokal and C. Michener. *A Statistical Method for Evaluating Systematic Relationships*. University of Kansas science bulletin. University of Kansas, 1958.

[51] T. Sun, C. Shu, F. Li, H. Yu, L. Ma, and Y. Fang. An efficient hierarchical clustering method for large datasets with map-reduce. In *2009 International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 494–499, 2009.

[52] O. Tanaseichuk, A. Hadj Khodabakhshi, D. Petrov, J. Che, T. Jiang, B. Zhou, A. Santrosyan, and Y. Zhou. An efficient hierarchical clustering algorithm for large datasets. *Austin Journal of Proteomics, Bioinformatics*, 2(1):1–6, 2015.

[53] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[54] F. Wang, J. Li, J. Tang, J. Zhang, and K. Wang. Name disambiguation using atomic clusters. In *Proc. WAIM*, pages 357–364, 2008.

[55] Y. Wang, V. Narasayya, Y. He, and S. Chaudhuri. An efficient partition-based distributed agglomerative hierarchical clustering algorithm for deduplication.

Technical report, https://www.microsoft.com/en-us/research/publication/tech-report-pack/, 2021.

[56] C.-H. Wu, S.-J. Horng, and H.-R. Tsai. Efficient parallel algorithms for hierarchical clustering on arrays with reconfigurable optical buses. *J. Parallel Distrib. Comput.*, 60(9):1137–1153, Sept. 2000.

[57] W. Wu, C. Yu, A. Doan, and W. Meng. An interactive clustering-based approach to integrating source query interfaces on the deep web. In *Proc. SIGMOD*, page 95–106, 2004.

[58] H. Yin, A. R. Benson, J. Leskovec, and D. F. Gleich. Local higher-order graph clustering. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, page 555–564, New York, NY, USA, 2017. Association for Computing Machinery.

[59] V. Zappala, A. Cellino, P. Farinella, and Z. Knezevic. Asteroid families. i-identification by hierarchical clustering and reliability assessment. *The Astronomical Journal*, 100:2030–2046, 1990.

[60] W. Zhang, G. Zhang, X. Chen, Y. Liu, X. Zhou, and J. Zhou. Dhc: A distributed hierarchical clustering algorithm for large datasets. *Journal of Circuits, Systems and Computers*, 28(04):1950065, 2019.

[61] W. Zhang, G. Zhang, Y. Wang, Z. Zhu, and T. Li. Nnb: An efficient nearest neighbor search method for hierarchical clustering on large datasets. In *IEEE ICSC 2015*, pages 405–412, 2015.

[62] Y. Zhao, G. Karypis, and U. Fayyad. Hierarchical clustering algorithms for document datasets. *Data mining and knowledge discovery*, 10(2):141–168, 2005.

A Near-Optimal Approach to Edge Connectivity-Based Hierarchical Graph Decomposition

Lijun Chang
The University of Sydney
Lijun.Chang@sydney.edu.au

Zhiyi Wang
The University of Sydney
zwan9517@uni.sydney.edu.au

ABSTRACT

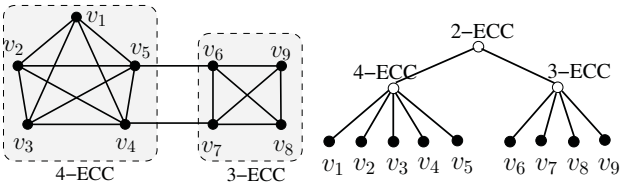
Driven by applications in graph analytics, the problem of efficiently computing all k-edge connected components (k-ECCs) of a graph G for *a user-given* k has been extensively and well studied. It is known that the k-ECCs of G for *all possible values of* k form a hierarchical structure. In this paper, we study the problem of efficiently constructing the hierarchy tree for G which compactly encodes the k-ECCs for all possible k values in space linear to the number of vertices n. All existing approaches construct the hierarchy tree in $O\big(\delta(G) \times \mathsf{T}_{\mathsf{KECC}}(G)\big)$ time, where $\delta(G)$ is the degeneracy of G and $\mathsf{T}_{\mathsf{KECC}}(G)$ is the time complexity of computing all k-ECCs of G for a specific k value. To improve the time complexity, we propose a divide-and-conquer approach running in $O\big((\log \delta(G)) \times \mathsf{T}_{\mathsf{KECC}}(G)\big)$ time, which is optimal up to a logarithmic factor. However, a straightforward implementation of our algorithm would result in a space complexity of $O((m+n)\log \delta(G))$. As main memory also becomes a scarce resource when processing large-scale graphs, we further propose techniques to optimize the space complexity to $2m + O(n \log \delta(G))$, where m is the number of edges in G. Extensive experiments on large real graphs and synthetic graphs demonstrate that our approach outperforms the state-of-the-art approaches by up to 28 times in terms of running time, and by up to 8 times in terms of main memory usage. As a by-product, we also improve the space complexity of computing all k-ECCs for a specific k to $2m + O(n)$.

PVLDB Reference Format:
Lijun Chang and Zhiyi Wang. A Near-Optimal Approach to Edge Connectivity-Based Hierarchical Graph Decomposition. PVLDB, 15(6): 1146 - 1158, 2022.
doi:10.14778/3514061.3514063

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://lijunchang.github.io/ECo-Decompose/.

1 INTRODUCTION

Graphs have been widely used to model the relationships among entities in real-world applications — such as social networks, collaboration networks, communication networks, E-commerce networks, web search, and biology — where entities are represented by vertices and relationships are represented by edges. With the proliferation of graph data, one of the fundamental problems in graph analytics is to compute the set of all *maximal* k-edge connected subgraphs, called *k-edge connected components* and abbreviated as k-ECCs, for a user-given k [4, 10, 35, 38]. A graph is *k-edge connected*, if it remains connected after removing any set of $k-1$ edges. For example, for the graph in Figure 1(a), the subgraphs induced by vertices $\{v_1, \ldots, v_5\}$ and $\{v_6, \ldots, v_9\}$ are the two 3-ECCs, while the former is also a 4-ECC. Computing k-ECCs has many applications, such as discovering cohesive blocks (communities) in social networks (*e.g.*, Facebook) [34], identifying closely related entities for social behavior mining [3], measuring robustness of communication networks [10], and matrix completability analysis [12].



(a) A toy graph (b) ECo-decomposition

Figure 1: A toy graph and its ECo-decomposition

Specifying the appropriate k value for an application is however not trivial and usually requires a trial-and-error process. Moreover, different applications may specify different k values. Thus, it is essential to pre-compute a data structure, such that k-ECCs for any given k can be efficiently retrieved from the data structure. It is known that the k-ECCs for all possible values of k form a hierarchical structure [37], as the k-ECCs for a specific k are disjoint and each k-ECC is entirely contained in a $(k-1)$-ECC [7]. For example, Figure 1(b) depicts the *hierarchy tree* $\mathcal{T}$ for the k-ECCs of the graph G in Figure 1(a), where leaf nodes are vertices of G and non-leaf nodes correspond to k-ECCs of G. With the constructed tree $\mathcal{T}$, the set of k-ECCs for any k can be extracted from $\mathcal{T}$ in time linear to the size of the k-ECCs. Thus, it becomes a problem of efficiently constructing the hierarchy tree for k-ECCs of all possible k values. We term the problem as *Edge Connectivity-based hierarchical graph decomposition*, abbreviated as ECo-*decomposition*.

Besides inheriting all the above applications, computing ECo-decomposition (*i.e.*, the hierarchy tree) also has a wide range of other applications as follows.

- *Hierarchical Organization and Visualization of Graphs.* ECo-decomposition constructs a hierarchical organization of a graph. It can facilitate graph-topology analysis [6], and assist users to visualize a graph in a multi-granularity manner [24], *i.e.*, zoom in and zoom out based on the edge connectivities of subgraphs.
- *Graph Sparsification.* ECo-decomposition efficiently computes the steiner connectivity for all edges (see Section 4.1).

Proceedings of the VLDB Endowment, Vol. 15, No. 6 ISSN 2150-8097.
doi:10.14778/3514061.3514063

It is shown in [5, 17] that independently sampling edges according to their steiner connectivities can sparsify a graph (*i.e.*, reduce the number of edges) while preserving the values of all cuts with a small multiplicative error.

- *Steiner Component Search.* ECo-decomposition is also an inherent preprocessing step towards efficient online steiner component search [7, 21], which is the problem of computing the subgraph with the maximum edge connectivity for a user-given set of query vertices [7].

The state-of-the-art approaches compute the ECo-decomposition (*i.e.*, construct the hierarchy tree $\mathcal{T}$) either in a top-down manner [7] or a bottom-up manner [37]. The top-down approach ECo-TD constructs the hierarchy tree by computing k-ECCs of G for all possible k values in increasing order [7], while the bottom-up approach ECo-BU computes k-ECCs of G for all possible k values in decreasing order [37]. Computation sharing techniques are exploited in ECo-TD and ECo-BU based on the observation that the working graph in an iteration for computing k-ECCs could be smaller than the input graph G, *e.g.*, the working graph in ECo-TD for computing k-ECCs is not G but the set of $(k-1)$-ECCs of G which are the results of the previous iteration [7]. Nevertheless, the worst-case time complexities of ECo-TD and ECo-BU are still $O\big(\delta(G) \times \mathsf{T}_{\mathsf{KECC}}(G)\big)$, where $\mathsf{T}_{\mathsf{KECC}}(G)$ is the time complexity of computing all k-ECCs of G for a specific k and $\delta(G)$ is the *degeneracy* of G which is equal to the maximum value among the minimum vertex degrees of all subgraphs of G [23]. It is interesting to observe that this time complexity is the same as the straightforward approach that *independently* computes k-ECCs of G for all possible k values, as the largest k will be no larger than $\delta(G)$.

Our Near-Optimal Approach. In this paper, we separate the computation into two parts: we first compute the steiner connectivity for all edges of G, and then construct the hierarchy tree $\mathcal{T}$ based on the computed steiner connectivities. The *steiner connectivity* of an edge (u, v), denoted as $sc(u, v)$, is the largest k such that a k-ECC of G contains (u, v). We show in the paper that the hierarchy tree of the ECo-decomposition can be constructed in $O(m)$ time given the steiner connectivities of all edges of G, where m is the number of edges of G. As a result, the main problem of ECo-decomposition is to efficiently compute the steiner connectivity for all edges of G.

We propose a divide-and-conquer approach ECo-DC to compute the steiner connectivities of all edges. The general idea is that given the set E_L^H of edges of G whose steiner connectivities are in the range $[L, H]$, *i.e.*, $E_L^H = \{(u, v) \in E(G) \mid L \leq sc(u, v) \leq H\}$, we compute the exact steiner connectivity for all edges of E_L^H as follows. If $L = H$, then $sc(u, v) = L$ for every edge $(u, v) \in E_L^H$ and the problem is solved. Otherwise, let $M = \lceil \frac{L+H}{2} \rceil$, we divide the problem into two sub-problems, E' and E'', to be solved recursively; here, $E' = E_L^{M-1} = \{(u, v) \in E(G) \mid L \leq sc(u, v) \leq M-1\}$ and $E'' = E_M^H = \{(u, v) \in E(G) \mid M \leq sc(u, v) \leq H\}$. The critical procedure is to efficiently divide a search problem E_L^H into two: E' and E''. We prove that E' is exactly the set of edges of E_L^H that are not in M-ECCs of the subgraph of G induced by E_L^H and all edges of G whose steiner connectivities are larger than H, and $E'' = E_L^H \backslash E'$. In addition, computation sharing techniques are exploited to bound the time complexity of ECo-DC by $O\big((\log \delta(G)) \times \mathsf{T}_{\mathsf{KECC}}(G)\big)$.

ECo-DC is optimal up to a logarithmic factor in terms of time complexity, since the time complexity of an ECo-decomposition algorithm is clearly lower bounded by $\mathsf{T}_{\mathsf{KECC}}(G)$. However, a naive implementation of ECo-DC would result in a space complexity of $O\big((n+m) \log \delta(G)\big)$ which is infeasible for large graphs. We first show that the space complexity can be reduced to $O(m+n \log \delta(G))$. Although this is much lower than the naive implementation, it is still too high to be applied to billion-scale graphs due to running out-of-memory, as the constant hidden by the big-O notation is large. In view of this, we further propose techniques to reduce the space complexity to $2m + O(n \log \delta(G))$ by explicitly bounding the constant on m by 2, while not increasing the time complexity; our space-optimized approach is denoted as ECo-DC-AA.

Extensive empirical studies on large graphs demonstrate that our approach ECo-DC-AA outperforms the state-of-the-art approaches ECo-TD and ECo-BU by up to 28 times in terms of running time, and by up to 8 times in terms of memory usage. Take the Twitter graph that has 1.2 billion undirected edges as an example, ECo-DC-AA finishes in 78 minutes by consuming 15GB memory, while ECo-TD and ECo-BU (as well as ECo-DC) run out-of-memory on a machine with 128GB memory; on the other hand, our space-optimized versions of ECo-TD and ECo-BU finish in 13.9 and 36.8 hours, respectively.

Our main contributions are summarized as follows.

- We propose a near-optimal approach to ECo-decomposition, which reduces the time complexity from $O(\delta(G) \times \mathsf{T}_{\mathsf{KECC}}(G))$ to $O((\log \delta(G)) \times \mathsf{T}_{\mathsf{KECC}}(G))$.
- We propose techniques to reduce the space complexity of our approach from $O\big((n+m) \log \delta(G)\big)$ to $2m + O(n \log \delta(G))$, such that billion-scale graphs can be processed in the main memory of a commodity machine.
- As a by-product, we significantly reduce the memory usage of the state-of-the-art k-ECC computation algorithm proposed in [10]. Moreover, our space optimization techniques can be generally applied to other graph algorithms.
- We conduct extensive empirical studies on large real and synthetic graphs to evaluate the efficiency of our approaches.

Organization. The rest of the paper is organized as follows. Section 2 gives preliminaries of the studied problem, and Section 3 presents the existing algorithms. We propose a near-optimal approach in Section 4, and develop techniques to reduce the memory usage of our algorithms in Section 5. Section 6 reports the results of our experimental studies, and Section 7 provides an overview of related works. Finally, Section 8 concludes the paper. Proofs are omitted due to limit of space and can be found in the full version [1].

2 PRELIMINARIES

In this paper, we consider a large *unweighted and undirected graph* $G = (V, E)$, with vertex set V and edge set E. The number of vertices and the number of *undirected* edges in G are denoted by $n = |V|$ and $m = |E|$, respectively. Given a vertex subset $V_s \subseteq V$, the subgraph of G induced by vertices V_s is denoted by $G[V_s] = (V_s, \{(u, v) \in E \mid u, v \in V_s\})$. Given an edge subset $E_s \subseteq E$, the subgraph of G induced by edges E_s is denoted by $G[E_s] = (\cup_{(u,v) \in E_s} \{u, v\}, E_s)$. For an arbitrary graph g, we use $V(g)$ and $E(g)$ to, respectively, denote its set of vertices and its set of edges.

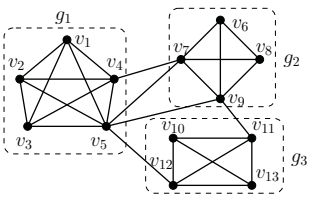


Figure 2: An example graph

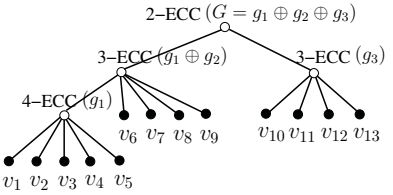


Figure 3: Hierarchy tree $\mathcal{T}$

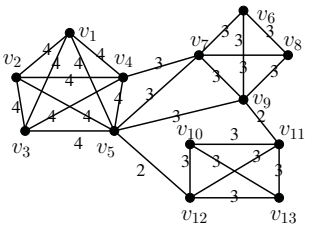


Figure 4: Steiner connectivities

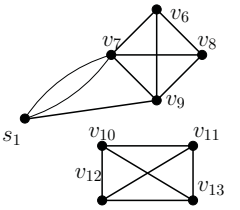


Figure 5: $\mathsf{GS}_3^3(G)$ for the graph in Figure 2

A graph is *k-edge connected* if the remaining graph is still connected after the removal of any $k-1$ edges from it. Note that, by definition, a graph with less than k edges (*e.g.*, consisting of a singleton vertex) is not considered to be k-edge connected. Then, k-edge connected component is defined as follows.

Definition 2.1: (k-edge Connected Component [10]) Given a graph G, a subgraph g of G is a *k-edge connected component* (abbreviated as k-ECC) of G if (i) g is k-edge connected, and (ii) g is *maximal* (*i.e.*, any super-graph of g is not k-edge connected).

Consider the graph in Figure 2, the entire graph is a 2-ECC but not a 3-ECC (since the graph will be disconnected after removing edges (v_5, v_{12}) and (v_9, v_{11})). The subgraph g_1 is a 4-ECC, and g_3 is a 3-ECC. Note that g_2, although is 3-edge connected, is not a 3-ECC since its super-graph $g_1 \oplus g_2$ is also 3-edge connected (*i.e.*, g_2 is not maximal). Here, $g_1 \oplus g_2$ denotes the union of g_1 and g_2, which also includes the cross edges between vertices of g_1 and vertices of g_2.

Hierarchy Tree of k-ECCs. It is shown in [7] that the k-ECCs of a graph satisfy the following properties.

(1) Each k-ECC is a vertex-induced subgraph.
(2) Any two distinct k-ECCs for the same k value are disjoint.
(3) Each k-ECC for $k > 1$ is entirely contained in a $(k-1)$-ECC.

Thus, the k-ECCs of a graph G for all possible k values can be compactly represented by a *hierarchy tree* $\mathcal{T}$, where leaf nodes of $\mathcal{T}$ correspond to vertices of G and non-leaf nodes of $\mathcal{T}$ correspond to distinct k-ECCs of G. Note that, to distinguish vertices of $\mathcal{T}$ from that of G, we refer to vertices of $\mathcal{T}$ as *nodes*. Figure 3 illustrates the hierarchy tree for k-ECCs of the graph in Figure 2.

We call non-leaf nodes of $\mathcal{T}$ as ECC nodes, and each ECC node is associated with a weight. An ECC node of weight k corresponds to a k-ECC which is the subgraph of G induced by all leaf nodes in the subtree of $\mathcal{T}$ rooted at the ECC node. For example, the left 3-ECC node in Figure 3 corresponds to the 3-ECC $g_1 \oplus g_2$ in Figure 2, which is the subgraph induced by vertices $v_1, \ldots, v_9$. Note that, if a subgraph g is both a k-ECC and a $(k+1)$-ECC, it is only represented once in the hierarchy tree by an ECC node of weight $k+1$. For example, the entire graph G is both a 2-ECC and a 1-ECC, and is represented by the ECC node of weight 2. Thus, each non-leaf node will have at least two children, and *the size of the hierarchy tree $\mathcal{T}$ is linear to n.*

It is worth pointing out that for any given k, the set of all k-ECCs of G can be efficiently obtained from the hierarchy tree $\mathcal{T}$ in time linear to the size of the k-ECCs.

Problem Statement. Given a large graph G, we study the problem of efficiently constructing the hierarchy tree for the set of all k-ECCs of G. We term this problem as *Edge Connectivity-based hierarchical graph decomposition*, and abbreviate it as ECo-*decomposition*.

In this paper, we will consider the algorithm for computing all k-ECCs of g for a given k as a black-box, denoted $\mathsf{KECC}(g, k)$. While any of the algorithms in [4, 10, 38] can be used to implement $\mathsf{KECC}(g, k)$, we implement the state-of-the-art algorithm in [10] in our experiments, and use $\mathsf{T}_{\mathsf{KECC}}(G)$ to denote the time complexity of $\mathsf{KECC}(g, k)$ when G is taken as the input graph.

3 EXISTING SOLUTIONS

In this section, we briefly review the two state-of-the-art approaches, and discuss their time complexities. The existing approaches compute the ECo-decomposition (*i.e.*, the hierarchy tree) either in a top-down manner [7] or in a bottom-up manner [37].

A Top-Down Approach: ECo-TD. The top-down approach constructs the hierarchy tree in a top-down manner, which is achieved by explicitly computing k-ECCs of G for all k values in increasing order [7]. The pseudocode is shown in Algorithm 1, denoted by ECo-TD. Initially, the root ECC node r of weight 1, which corresponds to the entire input graph G, is created for $\mathcal{T}$ (Line 1); note that, without loss of generality here G is assumed to be connected. Then, it recursively adds the set of children to each ECC node in a top-down fashion by invoking Construct-TD (Line 2).

Algorithm 1: ECo-TD(G)

```
1  Create the root ECC node r of T with weight 1;
2  Construct-TD(r, 1, G);
3  return T;

   Procedure Construct-TD(ecc, k, g)
4  φ_{k+1}(g) ← KECC(g, k+1);
5  if φ_{k+1}(g) is the same as g (i.e., g ∈ φ_{k+1}(g)) then
6      Change the weight of ecc to k+1;
7      Construct-TD(ecc, k+1, g);
8  else
9      for each vertex v of g that is not in subgraphs of φ_{k+1}(g) do
10         Create a leaf node for v to be a child of ecc in T;
11     for each connected subgraph g' ∈ φ_{k+1}(g) do
12         Create an ECC node ecc' of weight k+1 to be a child of
             ecc in T;
13         Construct-TD(ecc', k+1, g');
```

Given an ECC node ecc of weight k whose corresponding graph is g (*i.e.*, g is a k-ECC of G), Construct-TD constructs the set of

Algorithm 2: ECo-BU(G)

```
1  Create one leaf node in T for each vertex of G;
2  Compute an upper bound k̄_max(G) of the largest k such that G has
     a non-empty k-ECC;
3  for k ← k̄_max(G) down to 1 do
4      φ_k(G) ← KECC(G, k);
5      for each connected subgraph g ∈ φ_k(G) do
6          Create an ECC node ecc in T with weight k;
7          Add the set of nodes of T that correspond to vertices of g
             to be the children of ecc;
8          Contract g into a single super-vertex in G, to which ecc
             corresponds;
9  return T;
```

children of ecc. To do so, it first computes the set of $(k+1)$-ECCs of g (Line 4), denoted $\phi_{k+1}(g)$. If $\phi_{k+1}(g)$ is the same as g which means that g itself is $(k+1)$-edge connected (Line 5), then the weight of ecc is increased to $k+1$ (Line 6) and the recursion continues for g (Line 7). Otherwise, the set of children of ecc is added as follows: (i) a leaf node is added for each vertex of g that is not in $\phi_{k+1}(g)$ (Lines 9–10); (ii) an ECC node is added for each connected subgraph g' of $\phi_{k+1}(g)$ (Lines 11–12). The recursion continues for each newly added ECC node (Line 13).

A Bottom-Up Approach: ECo-BU. The bottom-up approach constructs the hierarchy tree in a bottom-up fashion, which is achieved by computing k-ECCs of G for all k values in decreasing order [37]. The pseudocode is shown in Algorithm 2, denoted ECo-BU.

Time Complexities. We first prove the following lemma.

Lemma 3.1: *Let $k_{\max}(G)$ be the largest k such that G contains a non-empty k-ECC, and $\delta(G)$ be the degeneracy of G which is equal to the maximum value among the minimum vertex degrees of all subgraphs of G [23]. Then, we have $k_{\max}(G) \le \delta(G)$.*

We actually observe that $k_{\max}(G) = \delta(G)$ for all real and synthetic graphs tested in our experiments. Thus, the largest k that is input to Construct-TD of Algorithm 1 is $\delta(G)$, and the time complexity of ECo-TD is $O\big(\delta(G) \times \mathsf{T}_{\mathsf{KECC}}(G)\big)$.[1] Note that, the time complexity analysis of ECo-TD is tight: for example, consider an input graph G that itself is $\delta(G)$-edge connected.

Following Lemma 3.1, the upper bound $\overline{k_{\max}}(G)$ can be set as $\delta(G)$ at Line 2 of Algorithm 2. Thus, the time complexity of ECo-BU is $O\big(\delta(G) \times \mathsf{T}_{\mathsf{KECC}}(G)\big)$,[2] as the degeneracy of G can be computed in $O(m)$ time [23]. Note that, the time complexity analysis of ECo-BU is also tight: for example, consider a graph that has no k-ECCs other than a $\delta(G)$-ECC and G itself which is 2-edge connected.

The degeneracy $\delta(G)$, although can be bounded by $O(\sqrt{m})$ in the worst case [31], may still be large, especially for large graphs. For example, $\delta(G)$ is more than 2,000 for the largest graphs tested in our experiments (see Table 1 in Section 6). As a result, ECo-BU and ECo-TD are taking excessively long time for processing large graphs due to their high time complexity of $O\big(\delta(G) \times \mathsf{T}_{\mathsf{KECC}}(G)\big)$, not to mention their high space complexity (see Section 5).

Handling Dynamic Graphs. Techniques for handling dynamic graphs have also been proposed in [7]. The general idea is based on the fact that deleting an edge from a graph or inserting a new edge into a graph will change the edge connectivity of the graph by at most 1, and moreover most of the k-ECCs will remain unchanged. These techniques can be directly adopted to maintain the hierarchy tree for dynamic graphs. We omit the details, as we focus on speeding up the construction of the hierarchy tree in this paper.

4 A NEAR-OPTIMAL APPROACH

In this section, we propose an approach for ECo-decomposition that runs in $O\big((\log \delta(G)) \times \mathsf{T}_{\mathsf{KECC}}(G)\big)$ time. To achieve this, we will need to avoid the explicit computation and enumeration of k-ECCs for all possible k values which would take $O\big(\delta(G) \times \mathsf{T}_{\mathsf{KECC}}(G)\big)$ time. Instead, we use a two-step paradigm, which first computes the steiner connectivity for all edges of G and then constructs the hierarchy tree based on the steiner connectivities, as follows.

```
1  Step-I: Compute the steiner connectivity for all edges of G;
2  Step-II: Construct the hierarchy tree based on the computed steiner
     connectivities;
```

In the following, we first in Section 4.1 propose an algorithm to compute the steiner connectivities of all edges in $O\big((\log \delta(G)) \times \mathsf{T}_{\mathsf{KECC}}(G)\big)$ time, and then in Section 4.2 present an algorithm to construct the hierarchy tree in $O(m)$ time based on the computed steiner connectivities.

4.1 Computing Steiner Connectivities

Definition 4.1: (Steiner Connectivity [7]) Given a graph G, the steiner connectivity of an edge (u, v), denoted $sc(u, v)$, is the largest k such that a k-ECC of G contains both u and v.

For example, in Figure 4, the steiner connectivity of each edge is computed as shown on the edge, *e.g.*, $sc(v_1, v_4) = 4$. Given a graph G, let $\phi_k(G)$ be the set of k-ECCs of G, then all edges of $\phi_k(G)$ have steiner connectivity at least k and all edges of G that are not in $\phi_k(G)$ have steiner connectivity smaller than k. In this subsection, we propose a divide-and-conquer approach for computing the steiner connectivities of all edges in a graph. Note that, although the concept of steiner connectivity is borrowed from [7], all our techniques in the following are new.

A Graph Shrink Operator $\mathsf{GS}_{k_1}^{k_2}(\cdot)$. We first introduce a graph shrink operator $\mathsf{GS}_{k_1}^{k_2}(\cdot)$ for $k_1 \le k_2$. Given a graph G, the result of $\mathsf{GS}_{k_1}^{k_2}(G)$ is still a graph. It is obtained from G by (1) removing all vertices and edges that are not in k_1-ECCs of G and (2) contracting each (k_2+1)-ECC of G into a super-vertex. Note that, the resulting graph of $\mathsf{GS}_{k_1}^{k_2}(\cdot)$ may have parallel edges. For example, $\mathsf{GS}_3^3(G)$ for the graph G in Figure 2 is shown in Figure 5 which is obtained by (1) removing edges (v_5, v_{12}) and (v_9, v_{11}), and (2) contracting

[1] Although the time complexity of ECo-TD is analyzed to be $O\big(\alpha(G) \times \mathsf{T}_{\mathsf{KECC}}(G)\big)$ in [7] where $\alpha(G)$ is the arboricity of G, this is the same as $O\big(\delta(G) \times \mathsf{T}_{\mathsf{KECC}}(G)\big)$ since $\alpha(G) \le \delta(G) \le 2\alpha(G) - 1$ [31].

[2] It is worth pointing out that the original algorithm in [37] is designed for I/O-efficient settings, and its time complexity cannot be bounded by $O\big(\delta(G) \times \mathsf{T}_{\mathsf{KECC}}(G)\big)$ as the upper bound $\overline{k_{\max}}(G)$ is set as the maximum degree of G in [37].

subgraph g_1 into a super-vertex s_1. There are two parallel edges between s_1 and v_7 in Figure 5.

The graph shrink operator $\mathtt{GS}_{k_1}^{k_2}(\cdot)$ has several properties which will be useful for computing steiner connectivities. Firstly, applying the operator $\mathtt{GS}_{k_1}^{k_2}(\cdot)$ preserves the steiner connectivity for all edges in the resulting graph.

Property 1: *Given a graph G and two integers $k_1 \le k_2$, the steiner connectivity of each edge of $\mathtt{GS}_{k_1}^{k_2}(G)$ when computed in $\mathtt{GS}_{k_1}^{k_2}(G)$ is the same as that computed in G.*

Secondly, the steiner connectivity for all edges of $\mathtt{GS}_{k}^{k}(G)$ is k. For example, all edges in Figure 5 have steiner connectivity 3.

Property 2: *Given a graph G and an integer k, every edge of $\mathtt{GS}_{k}^{k}(G)$ has steiner connectivity k.*

Thirdly, multiple operations of $\mathtt{GS}_{k_1}^{k_2}(\cdot)$ can be chained together.

Property 3: *Given a graph G and four integers $k_1 \le k_2$ and $k_3 \le k_4$ such that $\max\{k_1,k_3\} \le \min\{k_2,k_4\}$, we have $\mathtt{GS}_{k_3}^{k_4}\big(\mathtt{GS}_{k_1}^{k_2}(G)\big) = \mathtt{GS}_{\max\{k_1,k_3\}}^{\min\{k_2,k_4\}}(G)$.*

Our Divide-and-Conquer Approach: ECo-DC. From Property 2, we know that the steiner connectivities of all edges of $\mathtt{GS}_{k}^{k}(G)$ are k. Moreover, from the definitions of steiner connectivity and the graph shrink operator, we know that all edges whose steiner connectivities are k will be in $\mathtt{GS}_{k}^{k}(G)$. Thus, to compute steiner connectivities of all edges of G, it suffices to compute $\mathtt{GS}_{k}^{k}(G)$ for $k \in [1,\delta(G)]$. Instead of naively computing $\mathtt{GS}_{k}^{k}(G)$ independently for each $k \in [1,\delta(G)]$ which would take $O\big(\delta(G)\times \mathsf{T}_{\mathsf{KECC}}(G)\big)$ time, we propose a divide-and-conquer approach based on the fact that $\mathtt{GS}_{k}^{k}(G)$ is entirely contained in $\mathtt{GS}_{k_1}^{k_2}$ if $k_1 \le k \le k_2$.

Algorithm 3: ECo-DC(G)

```
1  Compute the degeneracy δ(G) of G;
2  Compute-DC(G, 1, δ(G));
3  ConstructHierarchy(G, sc(·,·));    /* See Algorithm 4 */;
4  return T;

   Procedure Compute-DC(g, L, H)
5  if L = H then
6      for each edge (u, v) ∈ E(g) do sc(u, v) ← L;
7  else
8      Choose an integer M such that L < M ≤ H;
9      φ_M(g) ← KECC(g, M);    /* Compute M-ECCs of g */;
10     Let g_1 be the graph obtained from g by contracting each
         connected subgraph of φ_M(g) into a super-vertex, and g_2 be
         φ_M(g);    /* g_1 = GS_L^{M-1}(G), g_2 = GS_M^H(G) */;
11     Compute-DC(g_1, L, M − 1);
12     Compute-DC(g_2, M, H);
```

The pseudocode of our approach is shown in Algorithm 3, denoted ECo-DC. It first computes the degeneracy $\delta(G)$ of G (Line 1), and then invokes procedure Compute-DC with input $(G,1,\delta(G))$ to compute the steiner connectivities of all edges (Line 2), while Line 3 constructs the hierarchy tree and will be discussed in Section 4.2. The input to Compute-DC consists of a graph g and an interval $[L,H]$. If $L=H$, then the steiner connectivities of all edges of g are set as L (Lines 5–6). Otherwise, an integer M is chosen such that $L<M\le H$ (Line 8), then the set $\phi_M(g)$ of M-ECCs of g is computed (Line 9) and two graphs g_1 and g_2 are obtained from g based on $\phi_M(g)$ (Line 10), and finally the algorithm continues on g_1 (Line 11) and on g_2 (Line 12).

We prove by the following lemma that when initially invoking Compute-DC with graph G and interval $[1,\delta(G)]$, the graph g being processed for each recursion with interval $[L,H]$ is $\mathtt{GS}_{L}^{H}(G)$.

Lemma 4.1: *For* Compute-DC*, if the input graph g is $\mathtt{GS}_{L}^{H}(G)$, then the two graphs g_1 and g_2 obtained at Line 10 are exactly $\mathtt{GS}_{L}^{M-1}(G)$ and $\mathtt{GS}_{M}^{H}(G)$, respectively.*

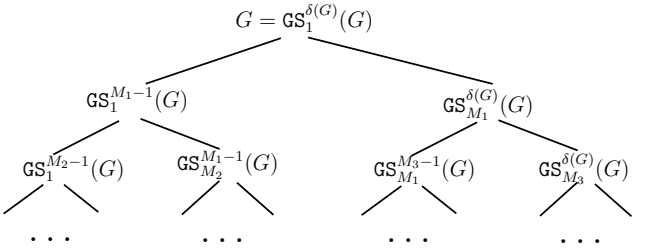


Figure 6: Recursion tree

Based on Lemma 4.1, the recursion tree of invoking Compute-DC with input $(G,1,\delta(G))$ is as shown in Figure 6.

The correctness and time complexity of Algorithm 3 are proved by the two theorems below.

Theorem 4.1: *Algorithm 3 correctly computes the steiner connectivity for all edges of G.*

Theorem 4.2: *The time complexity of Algorithm 3 is $O(h\times \mathsf{T}_{\mathsf{KECC}}(G))$, where h is the height of the recursion tree in Figure 6.*

Near-Optimal Time Complexity. Algorithm 3 correctly computes the steiner connectivities of all edges regardless of the choice of M at Line 8, as long as $L<M\le H$. Yet, the time complexity of Algorithm 3 would vary for different choices of M. For example, if M is always set as $L+1$ or always set as H, then the height of the recursion tree would be $\delta(G)$ and thus the time complexity of Algorithm 3 would be $O(\delta(G)\times \mathsf{T}_{\mathsf{KECC}}(G))$ on the basis of Theorem 4.2. To make the time complexity as low as possible, we will need to reduce the height of the recursion tree. Thus, we propose to set M as $\lceil \frac{L+H}{2} \rceil$, and prove in the following theorem that the time complexity of Algorithm 3 then becomes $O\big((\log\delta(G))\times \mathsf{T}_{\mathsf{KECC}}(G)\big)$.

Theorem 4.3: *By setting $M=\lceil \frac{L+H}{2} \rceil$, the time complexity of Algorithm 3 is $O\big((\log\delta(G))\times \mathsf{T}_{\mathsf{KECC}}(G)\big)$.*

Following the above theorem, we set $M=\lceil \frac{L+H}{2} \rceil$ in Algorithm 3. The time complexity of ECo-DC, which is $O\big((\log\delta(G))\times \mathsf{T}_{\mathsf{KECC}}(G)\big)$, is optimal up to a logarithmic factor $\log\delta(G)$. This is because the time complexity of ECo-decomposition cannot be lower than $\mathsf{T}_{\mathsf{KECC}}(G)$, as ECo-decomposition also implicitly computes the k-ECCs of G; specifically, the k-ECCs of G can be obtained from the hierarchy tree in time linear to the sizes of the k-ECCs.

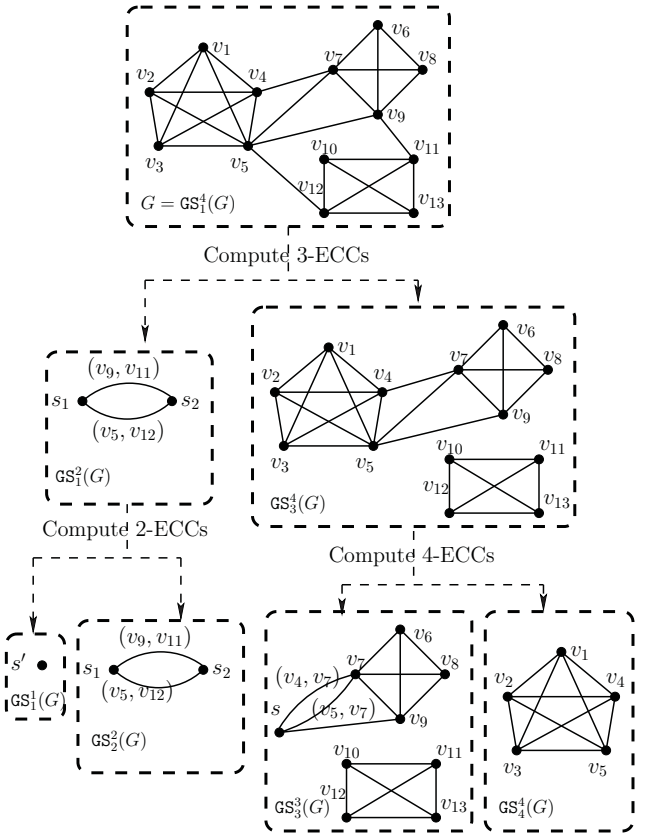


Figure 7: Running example of ECo-DC

Example 4.1: Here, we apply ECo-DC on the graph G in Figure 2 as an example. Figure 7 indicates the whole running process of ECo-DC on G, where the top-most part is G itself. The degeneracy is $\delta(G) = 4$. Then, we compute the steiner connectivities of all edges of G by invoking Compute-DC with input G and $[L, H] = [1, 4]$. Here, $\mathtt{GS}_1^4(G)$ is the same as G. As $L \neq H$ and $\lceil \frac{L+H}{2} \rceil = 3$, we compute the 3-ECCs of G and obtain the subgraphs induced by $S_1 = \{v_1, v_2, \ldots, v_9\}$ and $S_2 = \{v_{10}, \ldots, v_{13}\}$, respectively. Thus, we obtain the two graphs $\mathtt{GS}_1^2(G)$ and $\mathtt{GS}_3^4(G)$ as shown in the middle layer of Figure 7. The computation continues on these two graphs with intervals $[1, 2]$ and $[3, 4]$, respectively.

The graph $\mathtt{GS}_3^4(G)$ is composed of the two 3-ECCs of G as shown in right part of the middle layer of Figure 7. We compute the 4-ECCs of $\mathtt{GS}_3^4(G)$, and obtain the subgraph induced by vertices $\{v_1, v_2, \ldots, v_5\}$. Thus, all edges among vertices $\{v_1, v_2, \ldots, v_5\}$ have steiner connectivities 4 as indicated in $\mathtt{GS}_4^4(G)$, while the other edges have steiner connectivities 3 as demonstrated in $\mathtt{GS}_3^3(G)$.

The graph $\mathtt{GS}_1^2(G)$ is obtained by contracting each of S_1 and S_2 into a super-vertex as shown in the left part of the middle layer of Figure 7. In $\mathtt{GS}_1^2(G)$, there are two parallel edges between s_1 and s_2, corresponding to edges (v_9, v_{11}) and (v_5, v_{12}), respectively. As $\mathtt{GS}_1^2(G)$ is 2-edge connected, the steiner-connectivities of (v_9, v_{11}) and (v_5, v_{12}) are 2. □

4.2 Constructing the Hierarchy Tree

Given the steiner connectivities of all edges of a graph G, Algorithm 4 constructs the hierarchy tree of ECo-decomposition of G in a bottom-up manner. The main idea is as follows. First, the hierarchy tree $\mathcal{T}$ is initialized as a forest of singleton nodes. Then, for each edge $(u, v) \in E(G)$ in non-increasing order regarding $sc(\cdot, \cdot)$, we identify the tree in $\mathcal{T}$ (specifically, the root r_u of the tree) containing u and the tree (specifically, the root r_v of the tree) containing v. If u and v are already in the same tree (*i.e.*, $r_u = r_v$), then we do nothing. Otherwise, we merge the two trees into one in $\mathcal{T}$, with the root of this newly formed tree having weight $sc(u, v)$.

Algorithm 4: ConstructHierarchy

Input: A graph G with $sc(u, v)$ for each edge (u, v)
Output: The hierarchy tree of ECo-decomposition of G

```
1  Initialize an empty hierarchy tree T;
2  for each vertex u ∈ V(G) do Insert a singleton node u into T;
3  for each edge (u, v) ∈ E(G) in non-increasing sc(u, v) order do
4      Let r_u (resp r_v) be the root of the tree in T containing u (resp v);
5      if r_u = r_v then continue;
6      else if both r_u and r_v are ECC nodes with weight sc(u, v) then
7          Merge r_u and r_v into a single ECC node;
8      else if none of r_u or r_v is an ECC node with weight sc(u, v) then
9          Create a new ECC node in T with weight sc(u, v), and add
           r_u and r_v as its children;
10     else
11         Without loss of generality, assume r_u is an ECC node with
           weight sc(u, v), and add r_v as a child of r_u in T;
```

The pseudocode of constructing the hierarchy tree is illustrated in Algorithm 4, denoted by ConstructHierarchy. The input of the algorithm is a graph G with $sc(u, v)$ precomputed for each edge (u, v). It first initializes an empty hierarchy tree (Line 1), and creates a single-node tree in $\mathcal{T}$ for each vertex of G (Line 2). Then, the trees in $\mathcal{T}$ will be merged with each other to form ECC nodes in the hierarchy tree. For each edge $(u, v) \in E(G)$ sorted by $sc(u, v)$ in non-increasing order (Line 3), the roots of the trees in $\mathcal{T}$ containing node u and node v are found, represented by r_u and r_v respectively (Line 4). If $r_u = r_v$, it implies that vertices u and v have already been merged into the same tree so that the algorithm skips the current edge (Line 5); otherwise, the algorithm merges r_u and r_v into a single tree based on the following three cases. (1) If both r_u and r_v are ECC nodes with weight $sc(u, v)$, it merges r_u and r_v into a single ECC node (Lines 6–7). (2) If neither r_u nor r_v is an ECC node with weight $sc(u, v)$, it creates a new ECC node in $\mathcal{T}$ with weight $sc(u, v)$ whose children are r_u and r_v (Lines 8–9). (3) The last situation is that one of r_u or r_v is an ECC node with weight $sc(u, v)$ and the other is not; note that, if the other one is an ECC node, then its weight must be larger than $sc(u, v)$. Assume that r_u is the one with weight $sc(u, v)$, r_v would be added as a child of r_u. Similar steps would be applied to the situation where r_v is the one with weight $sc(u, v)$ (Lines 10–11).

The most time-consuming operation in Algorithm 4 is Line 4, which aims to find the root of the tree that contains a node u

in a forest $\mathcal{T}$. A naive implementation of this operation would take $O(n)$ time by tracing the parent pointers starting from node u in the tree, and then the total time complexity of Algorithm 4 would be $O(n \times m)$. This can be improved to $O(m)$ by resorting to the disjoint-set data structure. Recall that, a disjoint-set data structure $\mathcal{D}$ partitions a universe of elements into a collection of sets, and each set is represented by one of its element (called *representative*) [15]. There are two operations supported by the data structure $\mathcal{D}$: find the set that contains a specific element; merge two sets into one. In our case, the universe of the data structure $\mathcal{D}$ is the set of leaf nodes of the hierarchy tree $\mathcal{T}$, and there is a one-to-one correspondence between sets in $\mathcal{D}$ and trees in $\mathcal{T}$. Whenever we merge two trees in $\mathcal{T}$, we also union the two corresponding sets in $\mathcal{D}$. Moreover, we point each set (specifically, the representative element of the set) of $\mathcal{D}$ to the root of the tree in $\mathcal{T}$ to which the set corresponds. This pointer is used for efficiently identifying the root of the tree that contains a node (*i.e.*, Line 4). As each of the two operations on $\mathcal{D}$ takes amortized constant time [15] [3] and sorting the edges at Line 3 can be achieved in linear time by counting sort [15], the time complexity of Algorithm 4 is $O(m)$.

5 OPTIMIZING THE SPACE USAGE

A straightforward implementation of Algorithm 3 would result in a space complexity of $O((m+n)\log\delta(G))$, *i.e.*, each level of the recursion tree of Figure 6 would require storing a separate copy of the input graph G. This space complexity is too high for large graphs. In this section, we focus on optimizing the space usage of ECo-DC. We first in Section 5.1 discuss how to implement ECo-DC in $O(m+n\log\delta(G))$ space by using doubly-linked list-based graph representation, where the constant hidden by the big-O notation is large. Then, in Section 5.2 we further optimize the space usage of ECo-DC by using adjacency array-based graph representation and other nontrivial optimizations; this results in our space-efficient algorithm ECo-DC-AA that has space complexity $2m+O(n\log\delta(G))$. As a result, we can process billion-scale graphs with an ordinary PC. For example, experiments in Section 6 show that our adjacency array-based algorithms can process twitter-2010 and com-friendster, which have 1.2 and 1.8 billion undirected edges, respectively, with at most 15GB and 24GB main memory. In contrast, the linked list-based algorithms run out-of-memory even with 128GB memory.

5.1 Doubly-Linked List-based Implementation

In this subsection, we discuss how to implement ECo-DC by using doubly-linked list-based graph representation, which is also the representation used by the state-of-the-art KECC algorithm [10] and the two state-of-the-art ECo-decomposition algorithms [7, 37]. The main reason for the existing approaches to choose this representation is that KECC iteratively modifies the graph — *i.e.*, contract two (super-)vertices into one and remove (super-)vertices of degree less than k [10] — which can be easily implemented by using the linked list-based graph representation. We abstract these two graph modification operations as *vertex contraction* and *vertex removal*, respectively. Note that ECo-TD also uses the vertex removal operation (see Lines 11–13 of Algorithm 1), and ECo-BU uses the vertex contraction operation (see Line 8 of Algorithm 2).

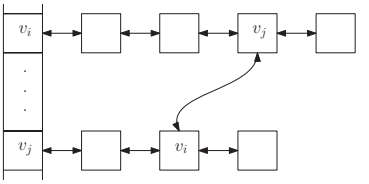


Figure 8: Doubly-linked list-based graph representation

Recall that, the linked list-based graph representation stores the adjacent edges of each vertex in a linked list [15]. For example, Figure 8 illustrates the linked lists for the adjacent edges of v_i and v_j. In addition, a cross pointer is constructed in the implementation for each edge (v_i, v_j) which points to its reverse direction (v_j, v_i), as each undirected edge will have two copies in the representation, one copy for each direction. Vertex removal can be implemented efficiently as follows. Suppose we are removing vertex v_i from the graph; note that we also need to remove all edges ending at v_i which scatter across the linked lists. To achieve this, we iterate through all the adjacent edges of v_i, and for each edge (v_i, v_j), we first locate its reverse edge (v_j, v_i) *via the cross pointer* and then remove (v_j, v_i) from the doubly-linked list of v_j which can be achieved in constant time. When it comes to vertex contraction, the process becomes slightly more complicated. Suppose we are contracting v_i and v_j. We use one of the vertices (*e.g.*, v_i) to represent the resulting super-vertex, and the process is divided into two parts: the edges starting from v_j should start from v_i; the edges end at v_j ought to end at v_i. For the first part, we could simply connect the head of the linked list of v_j to the tail of the linked list of v_i. For the second part, we iterate through all the adjacent edges of v_j, and for each edge (v_j, v_k), we first locate its reverse edge (v_k, v_j) via the cross pointer and then update the edge to be (v_k, v_i).

Based on the linked list-based graph representation, ECo-DC (*i.e.*, Algorithm 3) can be implemented fairly easily. Specifically, to construct $g_1 = \mathtt{GS}_L^{M-1}(g)$ and $g_2 = \mathtt{GS}_M^H(g) = \phi_M(g)$ from $g = \mathtt{GS}_L^H(G)$ at Line 10 of Compute-DC, we first split each linked list (that corresponds to the adjacent edges of a vertex) into two, one to be used in g_1 and the other in g_2, as g_1 and g_2 have disjoint sets of edges. We then apply the contraction operation for the edges in g_1. In this way, we do not create any new edges in Compute-DC; note however that, the number of vertices may double (*i.e.*, one copy in g_1 and one in g_2). Overall, ECo-DC has a space complexity of $O(m+n\log\delta(G))$, by noting that it traverses the recursion tree of Figure 6 in a depth-first manner.

5.2 Adjacency Array-based Implementations

Although the space complexity of ECo-DC has been reduced from $O((m+n)\log\delta(G))$ to $O(m+n\log\delta(G))$ in Section 5.1, this is still too high to be applied to large graphs (see our experimental results in Section 6) as the constant hidden by the big-O notation is large. Firstly, for each edge in the linked lists, three pointers and one number (where the number indicates the other end-point of the edge) need to be stored. Thus, the graph representation will consume at least $8m$ integers, by noting that each undirected edge

[3] To be more precise, the amortized time complexity of each operation on $\mathcal{D}$ is the inverse of the Ackermann function of n [15]. As this function grows very slowly and is bounded by 4 for all practical values of n, we consider it as a constant.

is stored twice. Secondly, the graph may be stored three times (*i.e.*, simultaneously have three copies in main memory) during the computation, *i.e.*, once in Compute-DC and twice in KECC as KECC will modify the graph that is input to it [10]. In this subsection, we propose an adjacency array-based implementation to explicitly bound the constant on m by 2 such that the space complexity becomes $2m+O(n\log\delta(G))$, and at the same time keep the time complexity unchanged which is challenging. Note that, we do not optimize the constant on $n\log\delta(G)$, as real-world graphs usually have much more edges than vertices, *i.e.*, m usually is the dominating factor.

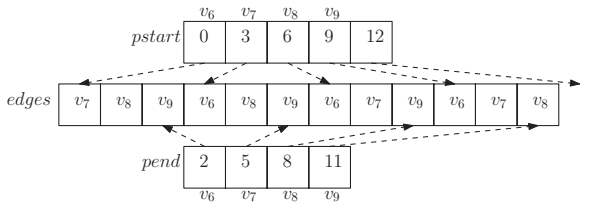


Figure 9: Adjacency array-based graph representation

The adjacency array-based graph representation is also known as the compressed sparse row (CSR) representation. It uses two arrays to represent a graph, and assumes that the vertices are taking ids from $\{0,\ldots,n-1\}$. We denote the two arrays by *pstart* and *edges*. The set of adjacent edges (specifically, neighbours) of each vertex is stored consecutively in an array, and then all such arrays are concatenated into the large array *edges*. The start position of the set of adjacent edges of vertex i in *edges* is stored in $pstart[i]$, and thus the set of adjacent edges of vertex i is stored consecutively in the subarray $edges[pstart[i],\ldots,pstart[i+1]-1]$. Figure 9 demonstrates such a representation for the subgraph g_2 of Figure 2; please ignore the part of "*pend*" for the current being. The array *pstart* is of size $n+1$, while the array *edges* is of size $2m$.

Efficient Implementation of Vertex Removal and Contraction. To achieve the space complexity of $2m+O(n\log\delta(G))$, we will not be allowed to create any new copies of *edges*, even if temporarily. This makes it challenging to efficiently implement vertex removal and vertex contraction which are the two primitive operations used by the algorithms. In the following, we discuss how to implement these two operations efficiently with the help of some additional data structures of size $O(n)$.

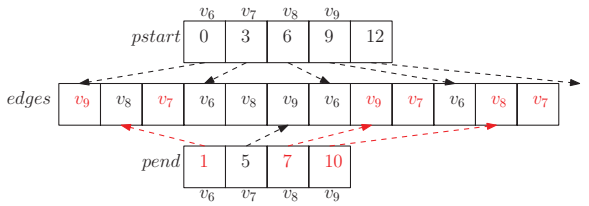
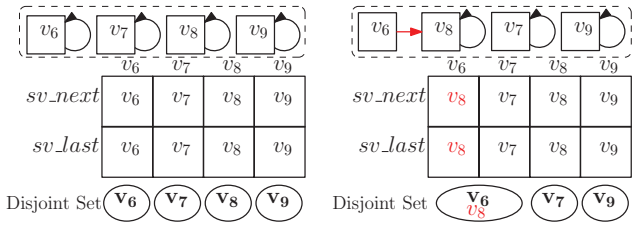


Figure 10: After removing vertex v_7

Vertex removal in the adjacency array-based graph representation can be implemented by marking the vertex as "removed". Recall that, when vertex i is "removed", the edge (j,i) that ends at i should also be removed from the adjacent edges of j for each neighbor j of i. This cannot be implemented efficiently without cross pointers, but *storing cross pointers is not affordable for achieving the space complexity of* $2m+O(n\log\delta(G))$. To circumvent this, we propose to remove (j,i) from the adjacent edges of j in a *lazy* way, *i.e.*, delay it to the moment when we actually need to traverse all adjacent edges of j. Thus, we introduce another array, named *pend*, of size n, where the entry $pend[j]$ explicitly stores the last position of the adjacent edges of vertex j in *edges* and is initialized with $pstart[j+1]-1$; see Figure 9. When we need to traverse all the adjacent edges of j, we loop through all the index values *idx* from $pstart[j]$ to $pend[j]$: if the edge $edges[idx]$ should have been removed (*i.e.*, the other end-point of this edge is "removed"), we first swap $edges[idx]$ with $edges[pend[j]]$ and then decrement $pend[j]$ by one. In this way, all the remaining (*i.e.*, active) adjacent edges of vertex j would be consecutive in *edges* starting from position $pstart[j]$ and ending at $pend[j]$, while the edges in *edges* whose indices are between $pend[j]+1$ and $pstart[j+1]-1$ are "removed". Thus, the amortized time of removing an edge is constant. For example, the result of removing vertex v_7 from the graph of Figure 9 is shown in Figure 10; here, for illustration purpose, we assume that the graph has been traversed once such that *edges* is reorganized.

When contracting vertex i and vertex j, following the same ideas as Section 5.1 we also use v_i to represent the resulting super-vertex and divide the process into two parts: the edges starting from v_j should start from v_i; the edges ending at v_j ought to end at v_i. For the first part, instead of moving adjacent edges around which would create temporary copies of *edges* and furthermore increase the time complexity, we use two additional arrays, *sv_next* and *sv_last*, each of size n to represent the super-vertices. That is, *sv_next* chains together all vertices that belong to the same super-vertex, implicitly represented as a singly-linked list; specifically, $sv\_next[i]$ stores the id of the next vertex (*i.e.*, after i) in the super-vertex. To efficiently merge two super-vertices (that are represented as singly-linked lists), we also store in $sv\_last[i]$ the id of the last vertex in the super-vertex i. For example, Figure 11(a) shows the values of *sv_next* and *sv_last* for the graph of Figure 9; note that, the part in the dotted rectangle illustrates the linked lists that represent the super-vertices, and is not physically stored. When contracting (super-)vertex i with (super-)vertex j, we first update $sv\_next[sv\_last[i]]$ to j to connect the two linked lists into one, and then update $sv\_last[i]$ to $sv\_last[j]$; this can be conducted in constant time. Note that, $sv\_last[i]$ is only useful and up-to-date if i is the first vertex in a linked list, *i.e.*, $sv\_last[\cdot]$ for all other vertices are not updated and will not be used. Figure 11(b) shows the result of contracting v_6 and v_8; notice that v_6 and v_8 are now linked together. To iterate over all edges adjacent to (super-)vertex i, we use a pointer p which is initialized as i and is then iteratively updated by $sv\_next[p]$ until reaching the end of the linked list. These p values correspond to ids of the vertices that are contracted into (super-)vertex i. Thus, the edges adjacent to (super-)vertex i are $edges[pstart[p],\ldots,pend[p]]$ for all p values along the iterations.

For the second part of vertex contraction (*i.e.*, edges ending at v_j ought to end at v_i), explicitly modifying the edge end-points without maintaining cross pointers would be time consuming. To tackle this issue, we propose to use an additional disjoint-set data structure of size $O(n)$ to represent the super-vertices. The universe of the data structure is the vertex set V, and each super-vertex corresponds to a set in the data structure that consists of the vertices contained in the



(a) Before contraction (b) After contraction

Figure 11: Example of contracting v_6 and v_8

super-vertex. When we contract two super-vertices, we also union their corresponding sets in the data structure. In addition, we point the representative of a set in the data structure to the vertex that represents the corresponding super-vertex, in the same way as that in constructing the hierarchy tree in Section 4.2. The last row of Figure 11 illustrates the disjoint sets, where the representative of a set is shown in bold, *e.g.*, v_6 and v_8 are in the same set in Figure 11(b) with v_6 being the representative.

Our Space-Optimized Algorithms. With the ideas presented above, we first optimize the space usage of KECC by using the adjacency array-based graph representation, as it is an essential procedure used in ECo-DC. We denote our space-optimized version of KECC as KECC-AA. Note that, with the above implementations of vertex removal and vertex contraction, the input graph to KECC-AA is always represented by *pstart* and *edge* which are not changed, although the order of the adjacent edges for each vertex may change. Thus, we do not need to store another copy of the input graph, and the space complexity of KECC-AA is $2m + O(n)$.

With KECC-AA, we are now ready to present our space-optimized version of ECo-DC. It is worth pointing out that directly replacing KECC with KECC-AA in Algorithm 3 will not achieve our desired space complexity. The main idea is still based on the fact that g_1 and g_2 in Algorithm 3 have disjoint sets of edges. But now, we cannot afford to first construct g_1 and g_2 from g, and then release the memory of g, as this will double the intermediate memory consumption. To tackle this issue, we always expand the right child of a node in the recursion tree (see Figure 6) before expanding the left child. This is based on the observation that, for a non-leaf node in the recursion tree, the graph processed by its right child is always a subgraph of the current graph while the graph processed by the left child is obtained by contracting each connected component (of the graph of the right child) into a super-vertex in the current graph. Thus, to process the right child, we can directly work on *pstart* and *edges* by rearranging the adjacent edges of each vertex and using a *local* array of size n to bookmark the number of adjacent edges of each vertex in the subgraph. After expanding the right child (and its descendants) and to process the left child, we further create a *local* copy of *sv_next*, *sv_last* and the disjoint-set data structure, which are all of size $O(n)$, to implement the contraction operation.

The pseudocode of the adjacency array-based implementation of ECo-DC is illustrated in Algorithm 5, denoted by ECo-DC-AA. It is generally similar to Algorithm 3, with three differences. Firstly, it invokes KECC-AA instead of KECC at Line 10. Secondly, it expands the right child first (Lines 11-13). Thirdly, it interleaves the execution of Algorithm 4 with Construct-DC-AA (Lines 2, 3, 7).

Algorithm 5: ECo-DC-AA(G)

```
1  Compute the degeneracy δ(G) of G;
2  Execute Lines 1–2 of Algorithm 4;
3  Construct-DC-AA(G, 1, δ(G));
4  return T;

   Procedure Construct-DC-AA(g, L, H)
5  if L = H then
6      for each edge (u, v) ∈ E(g) do
7          Execute Lines 4–11 of Algorithm 4 with sc(u, v) equal to L;
8  else
9      M ← ⌈(L+H)/2⌉;
10     φ_M(g) ← KECC-AA(g, M);
11     for each connected subgraph g' ∈ φ_M(g) do
12         Construct-DC-AA(g', M, H);
13         Contract g' into a super-vertex in g;
14     Construct-DC-AA(g, L, M − 1);
```

The reason of interleaving is that explicitly storing the steiner connectivities of all edges would increase the space consumption by at least $2m + O(n)$, and interleaving eliminates the requirement of storing the steiner connectivities. This interleaving is correct because the right child is always expanded before the left child for each node in the recursion tree (Figure 6), and thus the steiner connectivities are computed in non-increasing order. Note that, *we also exploit this interleaving to reduce the memory consumption for* ECo-DC, ECo-TD *and* ECo-BU *in our experiments.*

The correctness of ECo-DC-AA directly follows from the correctness of ECo-DC and the discussions in the above two paragraphs, and the time complexity of ECo-DC-AA remains the same as ECo-DC since our adjacency array-based implementation does not increase the time complexity of vertex removal and contraction. The space complexity of ECo-DC-AA becomes $2m + O(n \log \delta(G))$, as it conducts a depth-first traversal of the recursion tree (Figure 6) and each level of the recursion tree only requires a local data structure of size $O(n)$.

With the same idea as ECo-DC-AA, we can also implement ECo-TD and ECo-BU by using the adjacency array-based graph representation such that their space complexities become $2m+O(n)$ while their time complexities remain unchanged. We denote our space-optimized versions of ECo-TD and ECo-BU by ECo-TD-AA and ECo-BU-AA, respectively.

6 EXPERIMENTS

In this section, we conduct extensive performance studies to evaluate the efficiency and effectiveness of our techniques. Specifically, we evaluate the following ECo-decomposition algorithms:

- ECo-TD (Algorithm 1): the existing top-down approach proposed in [7] that uses the doubly-linked list-based graph representation.
- ECo-BU (Algorithm 2): an adaptation of the existing bottom-up approach proposed in [37] that uses the doubly-linked list-based graph representation.
- ECo-DC (Algorithm 3): our near-optimal approach (Algorithm 3) that uses the doubly-linked list-based graph representation and has a space complexity of $O(m + n \log \delta(G))$.

Table 1: Statistics of graphs ($\bar{d}$: average degree, δ: degeneracy)

ID	Dataset	m	n	$\bar{d}$	δ
D1	ca-CondMat	91,286	21,363	8.55	25
D2	soc-Epinions1	405,739	75,877	10.69	67
D3	web-Google	3,074,322	665,957	9.23	44
D4	as-Skitter	11,094,209	1,694,616	13.09	111
D5	cit-Patents	16,518,947	3,774,768	8.75	64
D6	soc-pokec	22,301,964	1,632,803	27.32	47
D7	wiki-topcats	25,444,207	1,791,489	28.41	99
D8	com-lj	34,681,189	3,997,962	17.35	360
D9	soc-LiveJournal1	42,845,684	4,843,953	17.69	372
D10	com-orkut	117,185,083	3,072,441	76.28	253
D11	uk-2002	261,556,721	18,459,128	28.34	943
D12	webbase	854,809,761	115,554,441	14.79	1,506
D13	twitter-2010	1,202,513,344	41,652,230	57.74	2,488
D14	com-friendster	1,806,067,135	65,608,366	55.06	304

- ECo-TD-AA, ECo-BU-AA and ECo-DC-AA: space-optimized versions of ECo-TD, ECo-BU and ECo-DC by using the adjacency array-based graph representation (Section 5.2).

In addition, we also evaluate two k-ECC computation algorithms:

- KECC: the state-of-the-art algorithm proposed in [10] that uses the doubly-linked list-based graph representation.
- KECC-AA: our space-optimized version of KECC that uses the adjacency array-based graph representation (Section 5.2).

All algorithms are implemented in C++ and compiled with GNU GCC with the -O3 optimization. All experiments are conducted on a machine with Intel(R) Xeon(R) 3.6GHz CPU and 128GB memory running Ubuntu. We evaluate the performance of all algorithms on both real and synthetic graphs as follows.

Real Graphs. We evaluate the algorithms on fourteen real graphs from different domains, which are downloaded from the Stanford Network Analysis Platform [4] and the Laboratory of Web Algorithmics [5]. Statistics of the graphs are shown in Table 1, where the second last column and the last column respectively show the average degree and the degeneracy. The graphs are ranked regarding their numbers of edges. We denote the graphs by D1, . . ., D14.

Synthetic Graphs. We evaluate the algorithms on power-law graphs that are generated by the graph generator GTGraph [6]. A power-law graph is a random graph in which edges are randomly added such that the degree distribution follows a power-law distribution. Firstly, we generate fourteen power-law graphs, PL1, . . ., PL14, where the number of vertices varies from 16 thousand to 133 million with an increasing factor of 2. The average degree of the power-law graphs are around 24.5; as a result, the number of undirected edges of the power-law graphs varies from 198 thousand to 1.6 billion. The degeneracy of these graphs varies from 18 to 25.

Secondly, we further generate six power-law graphs fixing the number of vertices to be the same as PL7 (*i.e.*, around one million), PL7_1, . . ., PL7_6, where the number of edges increases with a factor of 2. The resulting degeneracy of these graphs increases from 21 (for PL7) to 1, 380 (for PL7_6), also roughly with a factor of 2.

Evaluation Metrics. For all the evaluations, we record both the processing time and the peak main memory usage. Each testing is run three times, and the average results are reported. All algorithms are run in main memory and use a single thread. For the reported processing time, we exclude the I/O time that is used for loading a graph from disk to main memory. The peak memory usage of a program is recorded by /usr/bin/time [7].

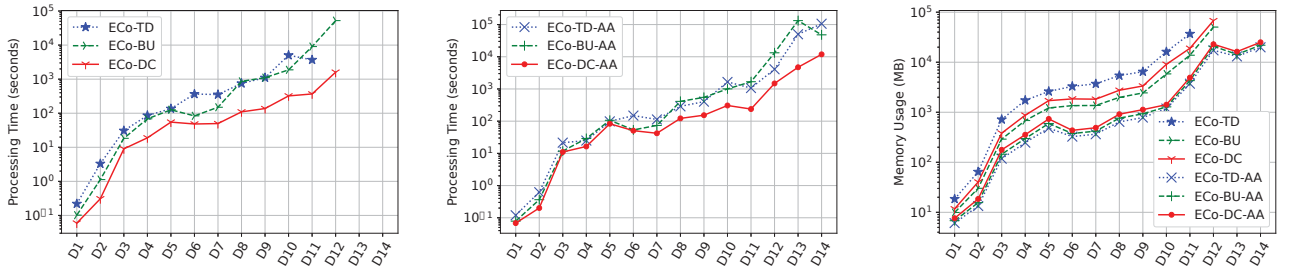
6.1 Results for ECo-decomposition

In this subsection, we evaluate the six ECo-decomposition algorithms regarding their processing time and main memory usage.

Results on Real Graphs. We first evaluate the algorithms on real graphs. The results are illustrated in Figure 12. For better comparison, we separate the algorithms into two groups: linked list-based algorithms (*i.e.*, ECo-TD, ECo-BU, and ECo-DC), and space-optimized algorithms (*i.e.*, ECo-TD-AA, ECo-BU-AA, and ECo-DC-AA). The processing time of the three linked list-based algorithms is illustrated in Figure 12(a). We can see that our near-optimal approach ECo-DC consistently runs faster than the two state-of-the-art approaches ECo-TD and ECo-BU, which is inline with our theoretical analysis that the former has a lower time complexity than the latter two. However, all the three algorithms fail to process the two billion-scale graphs D13 and D14, due to running out-of-memory. On the other hand, our space-optimized algorithms are able to process these billion-scale graphs as shown in Figure 12(b), due to their reduced main memory usage. The overall trend is similar to their counterparts in Figure 12(a), *i.e.*, ECo-DC-AA consistently performs the best. When comparing the top-down approach ECo-TD-AA with the bottom-up approach ECo-BU-AA, there is no clear winner despite of having the same time complexity, as their practical performance is sensitive to the graph topology. For example, the processing time of ECo-TD-AA, ECo-BU-AA, and ECo-DC-AA on D13 are respectively 13.9hrs, 36.8hrs and 1.3hrs, while that on D14 are respectively 29.4hrs, 13.3hrs and 3.3hrs.

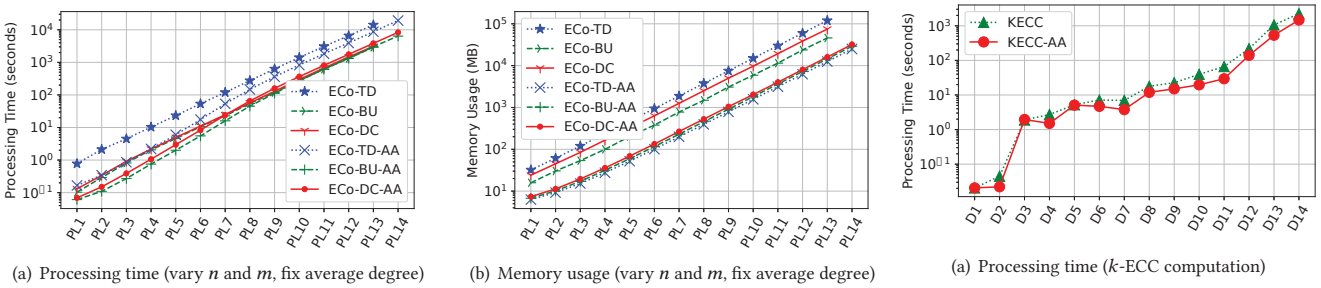
The main memory usage of the six algorithms is demonstrated in Figure 12(c). It is evident that our space-optimized algorithms (ECo-TD-AA, ECo-BU-AA, ECo-DC-AA) consume much less memory than the linked list-based algorithms (ECo-TD, ECo-BU, ECo-DC), where ECo-TD and ECo-BU are the two state-of-the-art approaches. For example, the peak memory usage of our space-optimized algorithms is at most 15GB for D13 and is at most 24GB for D14, while the linked list-based algorithms run out-of-memory even with 128GB memory. There are two things worth mentioning for Figure 12(c). Firstly, it appears that ECo-TD consumes more memory than ECo-DC. This is due to implementation differences, *i.e.*, we used the original implementation of ECo-TD from [7] while our implementations of ECo-BU and ECo-DC slightly optimized the constant on m in the space complexity. We do not optimize the code of ECo-TD, as linked list-based implementations, which are outperformed by their space-optimized counterparts, are not our main focus. Secondly, the linked list-based algorithm consume more memory on D13 than on D12, while for our space-optimized algorithms, the situation is the opposite. This is because (1) D13 has more edges but less vertices than D12, (2) the memory usage of linked list-based algorithms is mainly dominated by the part on

[4] http://snap.stanford.edu/
[5] http://law.di.unimi.it/datasets.php
[6] http://www.cse.psu.edu/~madduri/software/GTgraph/
[7] https://man7.org/linux/man-pages/man1/time.1.html

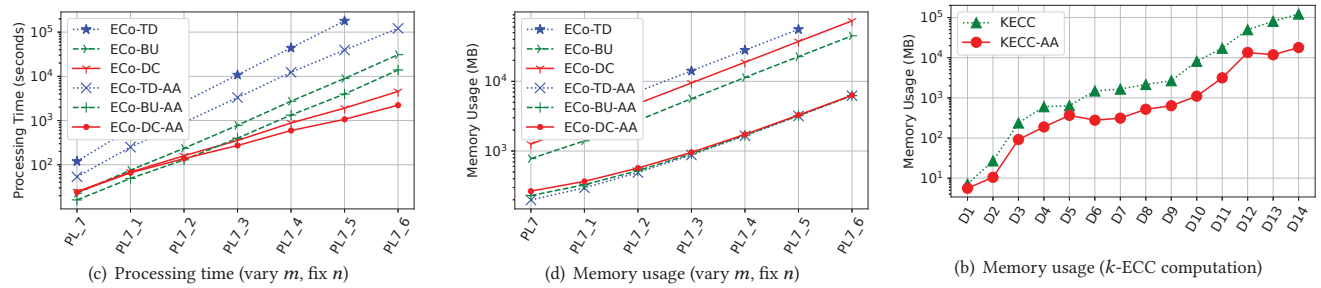


(a) Processing time of linked list-based algorithms (b) Processing time of space-optimized algorithms (c) Memory usage of all algorithms

Figure 12: Results of ECo-decomposition on real graphs (best viewed in color)



(a) Processing time (vary n and m, fix average degree) (b) Memory usage (vary n and m, fix average degree)



(c) Processing time (vary m, fix n) (d) Memory usage (vary m, fix n)

Figure 13: Results of ECo-decomposition on power-law graphs

(a) Processing time (k-ECC computation)

(b) Memory usage (k-ECC computation)

Figure 14: Results of k-ECC on real graphs

m in the space complexity, while the memory usage of our space-optimized algorithms is also affected by the part on n. This is also observed for k-ECC computation algorithms in Figure 14(b).

Results on Synthetic Graphs. The processing time and memory usage of the six algorithms on power-law graphs are shown in Figure 13. The overall trend is similar to that on real graphs in Figure 12. That is, our divide-and-conquer algorithms ECo-DC and ECo-DC-AA run the fastest, and our space-optimized algorithms consume much less memory than the linked list-based algorithms, *e.g.*, the latter run out-of-memory on PL14 which has 1.6 billion undirected edges. It is interesting to observe that our space-optimized bottom-up approach ECo-BU-AA also perform quite well on power-law graphs that have small degeneracy (*i.e.*, at most 25), see Figure 13(a). The results on power-law graphs by varying m and fixing n are shown in Figure 13(c) and Figure 13(d); note that the degeneracy of these graphs also increases with m. We can see that ECo-BU-AA now runs slower than ECo-DC-AA when the degeneracy becomes large, *e.g.*, the degeneracy of PL7_5 and PL7_6 are 705 and 1,380, respectively. From Figure 13, we can also observe that ECo-DC-AA scales almost linearly to large graphs for both the processing time and the memory usage.

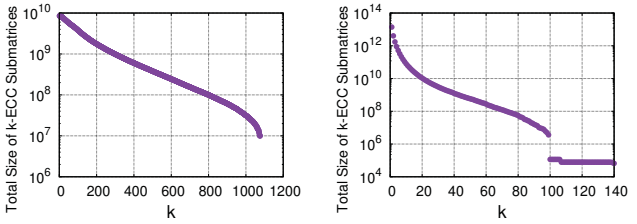
Table 2: Compare KECC-AA with NetworkX ($k = 8$)

Dataset	NetworkX		KECC-AA	
	Time (s)	Memory (MB)	Time (s)	Memory (MB)
D1	768.89	164.66	0.021	5.73
D2	1412.99	772.74	0.022	23.16

6.2 Results for k-ECC Computation

In this subsection, we evaluate our space-optimized algorithm KECC-AA for k-ECC computation. We first compare KECC-AA with the linked list-based counterpart KECC that is proposed in [10]. The results on real graphs for $k = 8$ are shown in Figure 14. We can observe that KECC-AA significantly reduces the memory usage compared with KECC. For example, KECC consumes 78GB and 119GB memory respectively for processing D13 and D14, while KECC-AA only consumes 11GB and 17GB memory for these two graphs. It is also interesting to see that KECC-AA is slightly faster than KECC. This is because KECC-AA benefits from increased cache hit-rate by using adjacency array-based graph representation.

We also compare KECC-AA with the k-ECC computation algorithm in NetworkX, a popular Python module for graph analytics. The results on the two smallest real graphs D1 and D2 for $k = 8$ are shown in Table 2; we do not test NetworkX on larger graphs as



(a) Netflix (b) Amazon_reviews

Figure 15: Matrix completability analysis

it is too slow. We can see that KECC-AA significantly outperforms NetworkX for k-ECC computation, *e.g.*, on D2, KECC-AA is more than 60,000 times faster and consumes 32 times less memory than NetworkX. Although there are factors of programming language difference (*i.e.*, C++ vs. Python), it is clear that KECC-AA has significant advantages over the implementation in NetworkX. It will be an interesting future work to implement KECC-AA in NetworkX.

6.3 Applications

In this subsection, we illustrate applying our ECo-decomposition algorithms in applications. Firstly, our algorithms directly speed up the index construction for steiner component search studied in [7, 21], which use the hierarchy tree as an index structure for efficiently processing online queries. Secondly, our algorithms can facilitate matrix completability analysis, where matrix completion is typically used for recommendation [12]. Specifically, a matrix can be represented as a bipartite graph $G = (U \cup L, E)$ with $U \cap L = \emptyset$ and $E \subseteq U \times L$. Each row of the matrix corresponds to a vertex of U, each column corresponds to a vertex of L, and each non-zero entry at position (i, j) corresponds to an undirected edge between $i \in U$ and $j \in L$. The problem of matrix completion is to predicate values for the entries of the matrix that currently have value 0 (*i.e.*, with value missing). It has been shown in [12] that the higher the edge connectivity of the corresponding bipartite graph, the more accurate the low-rank matrix completion. Thus, the higher the value of k such that i and j are contained in the same k-ECC, the more accurate the predicated value of the (i, j)-th entry of the matrix. The hierarchy tree constructed by our algorithms can be used to efficiently obtain the largest k such that i and j are contained in the same k-ECC, and thus to estimate the accuracy of the matrix completion for the (i, j)-th entry. Also, the hierarchy tree can be used to efficiently retrieve the submatrices, whose corresponding bipartite graphs are k-edge connected, to run the matrix completion algorithm and can be used to provide a guide on choosing the appropriate k. For example, Figures 15(a) and 15(b) show the total size of the submatrices whose corresponding bipartite graphs are k-edge connected, for datasets Netflix and Amazon_reviews; here, the size of a submatrix is #rows × #columns. Netflix [8] has $|U| = 480,189$, $|L| = 17,770$, $|E| = 100,480,507$, $k_{\max} = 1,076$, and Amazon_reviews [9] has $|U| = 6,643,669$, $|L| = 2,441,053$, $|E| = 29,928,296$, $k_{\max} = 140$. We can see that Netflix is much denser than Amazon_reviews and can be completed more accurately than Amazon_reviews. In particular, the total size of the submatrices whose corresponding bipartite graphs are 200-edge connected is more than 10% of the entire matrix size for Netflix, while there is no such submatrix for Amazon_reviews.

[8] https://www.kaggle.com/netflix-inc/netflix-prize-data
[9] http://snap.stanford.edu/data/web-Amazon-links.html

7 RELATED WORK

Besides the existing works on ECo-decomposition as discussed in Sections 1 and 3, we categorize other related works as follows.

***k*-ECC Computation.** In the literature, there are three approaches for computing all k-ECCs of a graph for a given k: cut-based approach [26, 35, 38], decomposition-based approach [10], and randomized approach [4]. In this paper, we adopted the decomposition-based approach [10] for k-ECC computation — which is the state of the art — and further optimized its memory usage.

Edge Connectivity Computation. Computing the edge connectivity between two vertices has been studied in graph theory [18], which is achieved by maximum flow techniques [15]. The state-of-the-art algorithms compute the maximum flow exactly in $O(n \times m)$ time [25] and approximately in almost linear time [22, 30]. Index structures have also been developed to efficiently process vertex-to-vertex edge connectivity queries [2, 20]. However, steiner connectivity as computed in this paper, which measures the connectivity in a *subgraph*, is different from edge connectivity as computed in [2, 20], which measures the connectivity in the *input graph*. Thus, these techniques cannot be applied. Moreover, it is worth mentioning that none of our algorithms involve maximum flow computation.

Cohesive Subgraph Computation. Extracting cohesive subgraphs from a large graph has also been extensively studied in the literature (see [9] for a recent survey). Here, the cohesiveness of a subgraph usually is measured by the minimum degree (aka, k-core) [29, 36], the average degree (aka, edge density) [8, 11, 19], the minimum number of triangles each edge participates in (aka, k-truss) [14, 27], and the vertex connectivity [33]. For some of the measures, the cohesive subgraphs for different cohesiveness values also form hierarchical structures and efficient algorithms have been proposed to construct these hierarchical structures, *e.g.*, core decomposition [13], truss decomposition and its higher-order variants [28], and density-friendly graph decomposition [16, 32]. However, due to inherently different problem natures, these techniques are inapplicable to computing ECo-decomposition of a graph.

8 CONCLUSION

In this paper, we proposed a near-optimal algorithm ECo-DC-AA for constructing the hierarchy tree of k-ECCs for all possible k values. ECo-DC-AA has both a lower time complexity and a lower space complexity compared with the state-of-the-art approaches ECo-TD and ECo-BU. Extensive experimental results on large graphs demonstrate that ECo-DC-AA outperforms ECo-TD and ECo-BU by up to 28 times in terms of running time and by up to 8 times regarding memory usage. As a result, ECo-DC-AA makes it possible to process billion-scale graphs in the main memory of a commodity machine. As a by-product, we also significantly reduced the memory usage of the state-of-the-art k-ECC computation algorithm.

ACKNOWLEDGMENTS

This work was supported by the Australian Research Council Fundings of FT180100256 and DP220103731.

REFERENCES

[1] [n.d.]. full version: https://lijunchang.github.io/pdf/2022-ecd-tr.pdf.

[2] Charu C. Aggarwal, Yan Xie, and Philip S. Yu. 2009. GConnect: A Connectivity Index for Massive Disk-resident Graphs. *PVLDB* 2, 1 (2009), 862–873.

[3] Rakesh Agrawal, Sridhar Rajagopalan, Ramakrishnan Srikant, and Yirong Xu. 2003. Mining newsgroups using networks arising from social behavior. In *Proc. of WWW'03*. 529–535.

[4] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. 2013. Linear-time enumeration of maximal K-edge-connected subgraphs in large networks by random contraction. In *Proc. of CIKM'13*. 909–918.

[5] András A. Benczúr and David R. Karger. 2002. Randomized Approximation Schemes for Cuts and Flows in Capacitated Graphs. *CoRR* cs.DS/0207078 (2002).

[6] Shai Carmi, Shlomo Havlin, Scott Kirkpatrick, Yuval Shavitt, and Eran Shir. 2007. A model of Internet topology using k-shell decomposition. *Proceedings of the National Academy of Sciences of the United States of America* 104, 27 (2007), 11150—-11154.

[7] Lijun Chang, Xuemin Lin, Lu Qin, Jeffrey Xu Yu, and Wenjie Zhang. 2015. Index-based Optimal Algorithms for Computing Steiner Components with Maximum Connectivity. In *Proc. of SIGMOD'15*.

[8] Lijun Chang and Miao Qiao. 2020. Deconstruct Densest Subgraphs. In *Proc. of WWW'20*. 2747–2753.

[9] Lijun Chang and Lu Qin. 2018. *Cohesive Subgraph Computation over Large Sparse Graphs*. Springer Series in the Data Sciences.

[10] Lijun Chang, Jeffrey Xu Yu, Lu Qin, Xuemin Lin, Chengfei Liu, and Weifa Liang. 2013. Efficiently computing k-edge connected components via graph decomposition. In *Proc. of SIGMOD'13*. 205–216.

[11] Moses Charikar. 2000. Greedy approximation algorithms for finding dense components in a graph. In *Proc. of APPROX'00*. 84–95.

[12] Dehua Cheng, Natali Ruchansky, and Yan Liu. 2018. Matrix completability analysis via graph k-connectivity. In *Proc. of AISTATS'18*. 395–403.

[13] James Cheng, Yiping Ke, Shumo Chu, and M. Tamer Özsu. 2011. Efficient core decomposition in massive networks. In *Proc. of ICDE'11*. 51–62.

[14] Jonathan Cohen. 2008. Trusses: Cohesive subgraphs for social network analysis. *National Security Agency Technical Report* (2008), 16.

[15] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. 2001. *Introduction to Algorithms*. McGraw-Hill Higher Education.

[16] Maximilien Danisch, T.-H. Hubert Chan, and Mauro Sozio. 2017. Large Scale Density-friendly Graph Decomposition via Convex Programming. In *Proc. of WWW'17*. 233–242.

[17] Wai Shing Fung, Ramesh Hariharan, Nicholas J. A. Harvey, and Debmalya Panigrahi. 2011. A general framework for graph sparsification. In *Proc. of STOC'11*. 71–80.

[18] Alan Gibbons. 1985. *Algorithmic Graph Theory*. Cambridge University Press.

[19] A. V. Goldberg. 1984. *Finding a Maximum Density Subgraph*. Technical Report. Berkeley, CA, USA.

[20] R. E. Gomory and T. C. Hu. 1961. Multi-Terminal Network Flows. *J. Soc. Indust. Appl. Math.* 9, 4 (1961). http://dx.doi.org/10.2307/2098881

[21] Jiafeng Hu, Xiaowei Wu, Reynold Cheng, Siqiang Luo, and Yixiang Fang. 2017. On Minimal Steiner Maximum-Connected Subgraph Queries. *IEEE Trans. Knowl. Data Eng.* 29, 11 (2017), 2455–2469.

[22] Jonathan A. Kelner, Yin Tat Lee, Lorenzo Orecchia, and Aaron Sidford. 2013. An Almost-Linear-Time Algorithm for Approximate Max Flow in Undirected Graphs, and its Multicommodity Generalizations. In *Proc. of SODA'13*.

[23] David W. Matula and Leland L. Beck. 1983. Smallest-Last Ordering and clustering and Graph Coloring Algorithms. *J. ACM* 30, 3 (1983), 417–427.

[24] An Nguyen and Seok-Hee Hong. 2017. k-core based multi-level graph visualization for scale-free networks. In *Proc. of PacificVis'17*. 21–25.

[25] James B. Orlin. 2013. Max flows in O(nm) time, or better. In *Proc. of STOC'13*. 765–774.

[26] Apostolos N. Papadopoulos, Apostolos Lyritsis, and Yannis Manolopoulos. 2008. SkyGraph: an algorithm for important subgraph discovery in relational graphs. *Data Min. Knowl. Discov.* 17, 1 (Aug. 2008), 20. https://doi.org/10.1007/s10618-008-0109-y

[27] Kazumi Saito and Takeshi Yamada. 2006. Extracting Communities from Complex Networks by the k-dense Method. In *Proc. of ICDMw'06*. 300–304.

[28] Ahmet Erdem Sariyüce and Ali Pinar. 2016. Fast Hierarchy Construction for Dense Subgraphs. *PVLDB* 10, 3 (2016), 97–108.

[29] Stephen B. Seidman. 1983. Network structure and minimum degree. *Social Networks* 5, 3 (1983), 269 – 287.

[30] Jonah Sherman. 2013. Nearly Maximum Flows in Nearly Linear Time. In *Proc. of FOCS'13*.

[31] Manuel Sorge and et al. 2013. The graph parameter hierarchy.

[32] Bintao Sun, Maximilien Danisch, T.-H. Hubert Chan, and Mauro Sozio. 2020. KClist++: A Simple Algorithm for Finding k-Clique Densest Subgraphs in Large Graphs. *Proc. VLDB Endow.* 13, 10 (2020), 1628–1640.

[33] Dong Wen, Lu Qin, Ying Zhang, Lijun Chang, and Ling Chen. 2019. Enumerating k-Vertex Connected Components in Large Graphs. In *Proc. of ICDE'19*. 52–63.

[34] Douglas R. White and Frank Harary. 2001. The cohesiveness of blocks in social networks: Node connectivity and conditional density. *Sociological Methodology* 31 (2001).

[35] Xifeng Yan, X. Jasmine Zhou, and Jiawei Han. 2005. Mining closed relational graphs with connectivity constraints. In *Proc. of KDD'05* (Chicago, Illinois, USA). 10. https://doi.org/10.1145/1081870.1081908

[36] Kai Yao and Lijun Chang. 2021. Efficient Size-Bounded Community Search over Large Networks. *Proc. VLDB Endow.* 14, 8 (2021), 1441–1453.

[37] Long Yuan, Lu Qin, Xuemin Lin, Lijun Chang, and Wenjie Zhang. 2017. I/O efficient ECC graph decomposition via graph reduction. *VLDB J.* 26, 2 (2017). https://doi.org/10.1007/s00778-016-0451-4

[38] Rui Zhou, Chengfei Liu, Jeffrey Xu Yu, Weifa Liang, Baichen Chen, and Jianxin Li. 2012. Finding Maximal k-Edge-Connected Subgraphs from a Large Graph. In *Proc. of EDBT'12*.

Hu-Fu: Efficient and Secure Spatial Queries over Data Federation

Yongxin Tong[†], Xuchen Pan[†], Yuxiang Zeng[‡], Yexuan Shi[†], Chunbo Xue[†], Zimu Zhou[#], Xiaofei Zhang[§], Lei Chen[‡], Yi Xu[†], Ke Xu[†], Weifeng Lv[†]

[†]State Key Laboratory of Software Development Environment, Beihang University, China
[†]Beijing Advanced Innovation Center for Future Blockchain and Privacy Computing, Beihang University, China
[‡]The Hong Kong University of Science and Technology [#]Singapore Management University [§]University of Memphis
[†]{yxtong, panxuchen, skyxuan, xuechunbo, xuy, kexu, lwf}@buaa.edu.cn, [‡]{yzengal, leichen}@cse.ust.hk,
[#]zimuzhou@smu.edu.sg, [§]xiaofei.zhang@memphis.edu

ABSTRACT

Data isolation has become an obstacle to scale up query processing over big data, since sharing raw data among data owners is often prohibitive due to security concerns. A promising solution is to perform secure queries over a federation of multiple data owners leveraging secure multi-party computation (SMC) techniques, as evidenced by recent federation work over relational data. However, existing solutions are highly inefficient on spatial queries due to excessive secure distance operations for query processing and their usage of general-purpose SMC libraries for secure operation implementation. In this paper, we propose Hu-Fu, the first system for efficient and secure spatial query processing on a data federation. The idea is to decompose the secure processing of a spatial query into as many plaintext operations and as few secure operations as possible, where fewer secure operators are involved and all secure operators are implemented dedicatedly. As a working system, Hu-Fu supports not only query input in native SQL, but also heterogeneous spatial databases (*e.g.*, PostGIS, Simba, GeoMesa, and SpatialHadoop) at the backend. Extensive experiments show that Hu-Fu usually outperforms the state-of-the-arts in running time and communication cost while guaranteeing security.

PVLDB Reference Format:
Yongxin Tong, Xuchen Pan, Yuxiang Zeng, Yexuan Shi, Chunbo Xue, Zimu Zhou, Xiaofei Zhang, Lei Chen, Yi Xu, Ke Xu, and Weifeng Lv. Hu-Fu: Efficient and Secure Spatial Queries over Data Federation. PVLDB, 15(6): 1159 - 1172, 2022.
doi:10.14778/3514061.3514064

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/BUAA-BDA/Hu-Fu.

1 INTRODUCTION

Efficient processing of spatial queries over large-scale data is essential for a wide spectrum of smart city applications including taxi-calling [68], logistics planning [65], map service [70], and contact-tracing [36] to name a few. Although the volume of spatial data continues to grow, it becomes increasingly difficult for these applications to take full advantage of the big spatial data due to the data isolation problem (*a.k.a.* isolated data) [9, 13, 49, 66]. Spatial datasets at city or nation scale are often privately possessed and separately owned by multiple parties, where sharing raw data among parties or uploading raw data to a third party (*e.g.*, a cloud) is prohibitive due to legal regulations (*e.g.*, GDPR [56]) or commercial reasons.

A promising paradigm to tackle the data isolation problem is to perform *secure* queries over a *data federation* [4], which consists of multiple data owners *a.k.a.* data silos [31, 36, 50], who agree on the same schema and manage their own data autonomously. Note that this paradigm differs from conventional federated databases [26, 48] in the extra security requirement. In general, secure query processing over data federation can be solved by well-known techniques such as secure multi-party computation (SMC) [7]. Yet, only recently did pioneer studies such as SMCQL [4] and Conclave [57] take the first step towards practice with efficient query execution plans upon SMC libraries for (relational) data federation. Unsurprisingly, more applications are being built on federations of spatial data owners.

Example 1. During COVID-19, several mobile network operators (*e.g.*, China Mobile [38] and China Telecom [53]) have cooperated as a spatial data federation to identify who has been to infectious areas through their location data [55]. Executing spatial queries (*e.g.*, range query or distance join) over a spatial data federation can help identify contacts in infectious areas across multiple organizations' spatial data without leaking privacy.

Example 2. AMAP (*a.k.a.* GaoDe Map) [3] has united over 8 Chinese travel companies into an integrated taxi-calling platform to offer users the taxis resources from all participating companies [54]. A spatial data federation can protect the distribution of taxis' locations of each company, which could be a business secret, from leaking to others.

Nevertheless, directly adapting the state-of-the-art data federation solutions [4, 57] to spatial data can be inefficient. From our empirical study (Sec. 2.2) of a kNN query on a real dataset, they are at least 142× slower, and have at least 1,216× higher communication cost than plaintext query processing. There are two reasons for such inefficiency. *(i)* Existing solutions process spatial queries with excessive secure distance operations, which occupy over 90% of the time cost. For example, SMCQL [4] and Conclave [57] would securely sort spatial objects by distances to the query point and pick the top-k objects, where each sorting involves numerous secure distance comparisons. *(ii)* Previous studies [4, 57] are built on

Proceedings of the VLDB Endowment, Vol. 15, No. 6 ISSN 2150-8097.
doi:10.14778/3514061.3514064

general-purpose SMC libraries, which may sacrifice the efficiency of specific operations for other considerations. For example, our experiment shows that the secure summation in ObliVM [35], the SMC library adopted by SMCQL [4], can be accelerated by 15× via dedicated implementations [19].

In this paper, we aim at efficient and secure spatial queries over a data federation, which we call as *federated spatial queries*. We mainly study five queries (federated range query/counting, kNN query, distance join and kNN join) commonly seen in spatial database research [16, 43, 64] and follow the semi-honest adversary model adopted by previous work [4, 23, 57, 60]. Moreover, we develop a more practical solution than [4, 57] by eliminating the need for an honest broker and supporting more data silos (these works support no more than 3 data silos whereas we tested up to ten).

To this end, we propose Hu-Fu, a system for efficient and secure processing of federated spatial queries. As explained above, secure operations are usually slow and easily become the efficiency bottleneck. Thus, the **key idea** of Hu-Fu is to decompose a federated spatial query into as many plaintext operations and as few secure operations as possible without compromising security, where *(i)* no secure distance-related operations are involved and *(ii)* the secure operations have implementations faster than those in general-purpose SMC libraries. To realize this idea and implement a practical system, Hu-Fu consists of three components: an query rewriter with novel decomposition plans, a set of drivers adaptable to heterogeneous databases and an easy-to-use query interface with SQL support. Specifically, the query rewriter identifies a set of plaintext and secure operators to cover the queries of interest, and adopts novel decomposition plans to minimize the usage of secure operators while ensuring security. The drivers provide the implementations of secure operators with dedicated SMC protocols and plaintext operations as interfaces on top of the heterogeneous spatial databases adopted by different data silos. The query interface supports spatial queries in native SQL for easy usage.

Our main contributions and results are summarized as follows.

- To the best of our knowledge, Hu-Fu is the first system on efficient and secure spatial queries over a data federation.
- We devise novel decomposition plans for federated spatial queries. After decomposition, an execution plan involves only a limited number of secure operators that can be effectively supported with fast and dedicated implementations.
- Hu-Fu is an efficient, easy-to-use system that supports query input in SQL and heterogeneous spatial databases, *e.g.*, PostGIS [45], MySQL [61], SpatiaLite [51], Simba [64], GeoMesa [27], and SpatialHadoop [16].
- Extensive evaluations show that Hu-Fu usually outperforms the state-of-the-arts [4, 57] in efficiency. Compared with two strong baselines, namely SMCQL-GIS and Conclave-GIS, which are extended from SMCQL [4] and Conclave [57] to spatial queries, Hu-Fu is up to 4 orders of magnitude faster and 5 orders of magnitude lower in communication than SMCQL-GIS and Conclave-GIS with the same security level.

In the rest of this paper, we define our problem scope and identify the inefficiency of existing solutions in Sec. 2. We present an overview of Hu-Fu in Sec. 3 and elaborate on its functional components in Sec. 4, Sec. 5 and Sec. 6. Finally, we present the evaluations in Sec. 7, review the related work in Sec. 8, and conclude in Sec. 9.

2 PROBLEM STATEMENT

This section clarifies our problem scope (Sec. 2.1) and highlights the challenges (Sec. 2.2) that motivate the design of Hu-Fu.

2.1 Problem Scope

We consider a data federation F ("federation" for short) consisting of n data silos ("silos" for short, denoted by F_i), where each silo holds multiple *spatial objects*. Each spatial object o has a location l_o and other attributes a_o. The federation supports *federated spatial queries* over the spatial objects of all silos under the following settings.

- *Spatial queries:* The federation F should support mainstream spatial queries including range query, range counting, kNN query, distance join, and kNN join [52, 64].
- *Autonomous databases:* Each silo is an autonomous database that does not share its raw spatial objects with other silos. This is aligned with real-world data federations [4–6, 57].
- *Semi-honest adversaries:* Each silo honestly executes queries received and returns authentic results, but may attempt to infer data from other silos during query execution. This assumption is common in query processing over a data federation [4, 57].

We focus on query processing with the following requirements.

- **Efficiency requirements.** We care about the *running time* and *communication cost* to execute *exact* queries over multiple silos. Short running time is desirable since real applications may process massive queries and a long latency can have bad effects (*e.g.*, it may cause an extended spread of diseases for contact tracing or degraded user experience for taxi-calling). Minimal communication cost is critical in distributed query processing [17, 43] and secure query processing [29]. Approximate query processing over data federation [6, 14] is out of our scope because applications such as contact tracing require accurate results. We consider multiple silos as aligned with real-world applications. Similar to existing federated query solutions [4, 57], the storage efficiency, which mainly depends on silos themselves, is not our primary concern.
- **Security requirements.** We target the scenario where input queries are public to all silos yet neither the query user nor any silo could deduce extra information from the final results. For instance, in federated kNN query, the query user can only know the final result (*i.e.* k nearest neighbors), and cannot infer the ownership of these k nearest neighbors. Such requirements are common in secure query processing [7].

Security is often of utmost priority due to laws (*e.g.*, GDPR [44] and CCPA [39]) on data protection. To satisfy the security requirement, existing systems [4, 57] rely on an honest broker to securely collect the partial answers (which may have sensitive data) from each silo. For other operations, they still rely on secure protocols (*e.g.*, summing the local counts from each silo to answer a range counting). However, real-world brokers (*e.g.*, Acxiom [1] and Experian [21]), which need to be paid a lot for data broker services, may still leak sensitive data for profit or by accident [47]. Thus, we do not assume an honest broker in Hu-Fu.

Formally, we define the federated spatial queries of interest below. They are common in existing spatial data systems [16, 43, 64]. Here, function $d(p, o)$ is the distance between spatial objects p and o.

Definition 1 (Federated Range Query/Counting)**.** Given a federation $F = \{F_1, \cdots, F_n\}$, and a query range $\mathcal{R}$, *federated range query* returns all objects $o \in F$ located in $\mathcal{R}$; *federated range counting* returns the number of these objects. These two queries should only return the final results without revealing any information of F_i (*e.g.*, the ownership of objects, the number of objects) to F_j $(j \neq i)$.

Definition 2 (Federated kNN Query)**.** Given a federation $F = \{F_1, \cdots, F_n\}$, a query point p and an integer k, *federated kNN query* returns a set $\text{kNN}(F, p, k)$ of k spatial objects such that

$$\forall o \in \text{kNN}(F, p, k), \forall o' \in F - \text{kNN}(F, p, k), d(p, o) \leq d(p, o')$$

without revealing information except the returned set to any F_i.

Definition 3 (Federated Distance Join)**.** Given an input dataset of spatial objects R, a federation $F = \{F_1, \cdots, F_n\}$ and a radius r, *federated distance join* returns each $o \in R$ with each $o' \in F$ that satisfies $d(o, o') \leq r$ as pairs, without revealing the ownership of $o' \in F_i$ to F_j $(j \neq i)$.

$$R \bowtie_r F = \{(o, o') | o \in R, o' \in F, d(o, o') \leq r\}$$

Definition 4 (Federated kNN Join)**.** Given an input dataset of spatial objects R, a federation $F = \{F_1, \cdots, F_n\}$ and k, *federated kNN join* returns each $o \in R$ with each $o' \in \text{kNN}(F, o, k)$ as pairs, without revealing information except the returned set to F_i.

$$R \bowtie_{\text{kNN}} F = \{(o, o') | o \in R, o' \in \text{kNN}(F, o, k)\}$$

2.2 Challenges

Federated spatial queries can be realized by secure multi-party computation (SMC) [7], as in prior studies for relational data [4, 57]. Nevertheless, our empirical study shows that they are highly inefficient on spatial queries, as explained below.

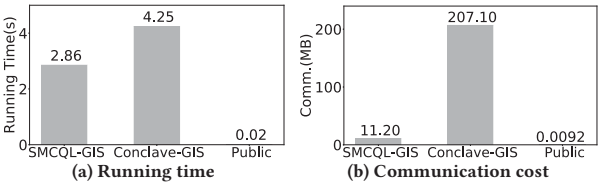
2.2.1 Inefficiency on Federated Spatial Queries. As an illustrative study, we perform federated kNN query by extending SMCQL [4] and Conclave [57], two representative solutions to secure query processing on (relational) data federations.

Overview of Existing Solutions. The general framework to apply SMC techniques for secure query processing over data federations is to decouple query execution into first *plaintext* queries within each silo and then *secure* computations of the final results across silos [4, 57]. This is because SMC protocols are slow and such a framework accelerates query processing without compromising security. Existing solutions differ in the underlying SMC techniques they apply for secure operations, where garbled circuit (GC) and secret sharing (SS) are two mainstream SMC techniques [7]. Specifically, SMCQL [4], the first solution for secure query processing over a data federation, uses ObliVM [35], a prevalent GC based library. Since ObliVM only supports two silos, Conclave [57] adopts an SS based technique (Sharemind [11]), which enables query processing on three silos.

Setup. We extend SMCQL [4] and Conclave [57] to federated kNN queries as follows. Following the "plaintext + secure" processing pipeline, each silo first conducts a plaintext kNN query and returns the k nearest points (along with their distances) to the query point. Then, the final k nearest neighbors are derived from these returned points, which are securely sorted by their distances to the query point and the k nearest ones are picked. We experiment with two silos with $k = 16$. Other implementations and experimental setup details are in Sec. 7.1.

Table 1: Percentage of time spent for *plaintext* or *SMC* operations for a federated kNN query via existing solutions.

System	Plaintext	SMC
SMCQL-GIS	0.14%	99.86%
Conclave-GIS	0.10%	99.90%



(a) Running time **(b) Communication cost**

Figure 1: Inefficiency of Conclave-GIS and SMCQL-GIS on federated kNN query, where SMCQL-GIS and Conclave-GIS are our extensions on SMCQL [4] and Conclave [57] to spatial queries (see Sec. 7.1).

Results. Fig. 1 plots the running time and communication cost to process a single federated kNN query leveraging existing solutions. The results are averaged over 50 queries. Compared with Public, *i.e.* plaintext kNN query execution without the security requirement, the secure counterpart incurs 142× to 212× longer running time and 1,216× to 22,510× higher communication cost. Although SMCQL-GIS yields a shorter running time and a lower communication cost than Conclave-GIS, SMCQL-GIS is *only applicable to the scenario of two silos* for its usage of GC based SMC techniques. Yet it still takes 2.86 seconds for a single federated kNN query, which can hurt user experiences in applications where time efficiency is critical.

2.2.2 Understanding the Efficiency Bottleneck. Prior studies are inefficient on federated spatial queries for the following reasons.

- **Excessive Secure Distance Operations**. When processing a federated kNN query, over 99% time is spent on SMC operations (*e.g.*, secure distance comparisons) as shown in Table 1. For example, SMCQL-GIS and Conclave-GIS adopt sorting to find k nearest neighbors among nk candidates by using $O(nk \log(nk))$ secure distance comparisons, and a single secure distance comparison takes 209 ms in SMCQL-GIS and 248 ms in Conclave-GIS, which equals to the time of at least 10^6 plaintext comparisons.
- **Reliance on General-Purpose Libraries.** Existing methods use general-purpose libraries to implement SMC operations (*e.g.*, ObliVM [35] in SMCQL [4]). General-purpose libraries sometimes sacrifice efficiency for generalization or compatibility. For example, the secure summation we used can be 16× faster than that in ObliVM (see Sec. 7). As will be shown in Sec. 4, we can process federated spatial queries with only a few secure operations. This facilitates acceleration with libraries dedicated to such operations [11, 19, 28].

Takeaways. Our study shows that existing secure query processing solutions (*e.g.*, [4, 57]) for data federations are inefficient for spatial queries. The inefficiency comes from *(i)* massive secure distance operations, and is exacerbated by *(ii)* adopting general-purpose libraries for these SMC operations. In response, we propose Hu-Fu, a solution with *(i)* a novel execution plan for federated spatial queries that involve notably fewer secure operations (see Sec. 4) and *(ii)* each secure operator can be implemented in high efficiency via dedicated algorithms (see Sec. 5). As next, we give an overview of Hu-Fu and elaborate on its functional modules in the following.

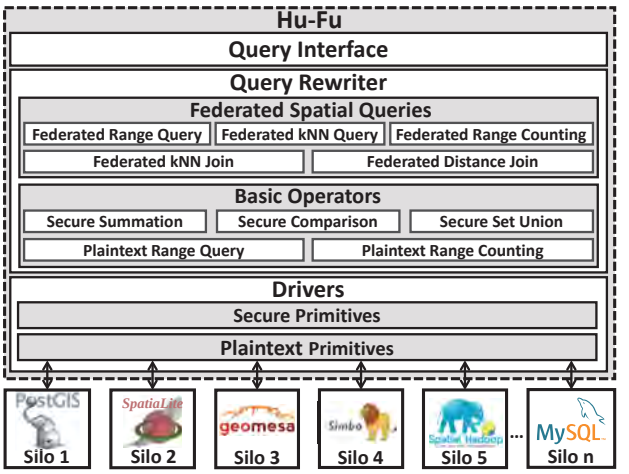


Figure 2: Illustration of Hu-Fu architecture.

3 HU-FU OVERVIEW

Hu-Fu is a solution that enables efficient and secure spatial queries over a data federation. It addresses the inefficiency of federated spatial query processing (see Sec. 2.2.2) via two modules: *(i)* a novel **query rewriter** that decomposes federated spatial queries into *plaintext and secure operators*, with the former executed within each silo and the latter across silos; *(ii)* **drivers** that implement these operators as *plaintext and secure primitives* leveraging dedicated algorithms and optimizations. Hu-Fu also contains a transparent **query interface** to support federated spatial queries written in native SQL. We briefly explain its architecture and workflow below.

3.1 Architecture

Fig. 2 illustrates the architecture of Hu-Fu, which consists of three modules: the query interface, the query rewriter and drivers. From a functional perspective, the query rewriter and drivers optimize the *efficiency* of federated spatial queries, and the query interface improves the *usability* of Hu-Fu.

Query Rewriter (Sec. 4). It decomposes federated spatial queries into plaintext operators (executed within silos) and secure operators (executed across silos). We define two *plaintext* operators (plaintext range query and range counting) and three *secure* operators (secure summation, comparison and set union) as the basic operators, upon which we design novel execution plans that decompose mainstream federated spatial queries (federated range query, range counting, kNN query, distance join and kNN join) into these basic operators.

Drivers (Sec. 5). Hu-Fu's drivers implement the basic operators defined in the query rewriter as efficient *primitives* that can adapt to heterogeneous spatial databases at the backend. Each operator is implemented by a specific primitive. Specifically, secure operators are implemented as *secure primitives* with dedicated optimizations [11, 19, 28]. Plaintext operators are implemented as *plaintext primitives* on top of the underlying spatial databases, which support various systems, *e.g.*, PostGIS [45], SpatiaLite [51], MySQL [61], GeoMesa [27], Simba [64] and SpatialHadoop [16].

Query Interface (Sec. 6). This module *(i)* provides a transparent and unified federation view to users, and *(ii)* supports federated spatial queries written in SQL. We implement the query interface by extending the schema manager and parser of Calcite [8]. We also provide interfaces such as JDBC for easy integration of Hu-Fu to users' programs.

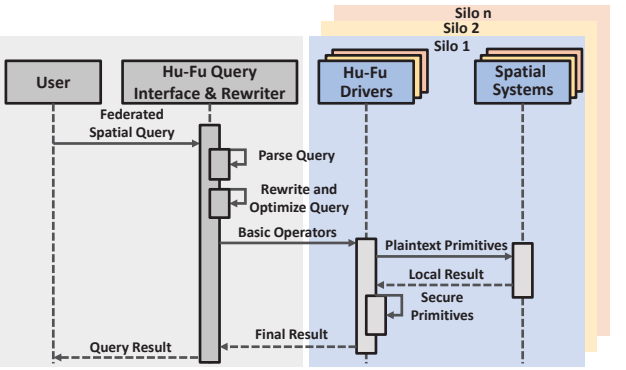


Figure 3: Illustration of Hu-Fu workflow.

3.2 Workflow

Fig. 3 shows the workflow of Hu-Fu with a user querying a data federation of n silos. The query interface and query rewriter are deployed on the user machine to provide a portal for spatial services. Each silo runs an instance of Hu-Fu drivers to interact with its underlying spatial databases.

Suppose the user's spatial service issues a federated spatial query written in SQL. When a federated spatial query comes in, it is first parsed by the query interface. Then the query rewriter transforms and optimizes the query into a sequence of plaintext and secure operators. These operators are then sent to drivers for execution as plaintext and secure primitives. First, the plaintext primitives are executed on the underlying spatial databases at each silo to get the local results. Afterward, the local results are collected to perform the secure primitives for the final query result, which is returned to the user by the query interface.

4 QUERY REWRITER

This section presents Hu-Fu's query rewriter, which decomposes federated spatial queries into multiple basic operators. We first define the basic operators in Sec. 4.1 before explaining the overall decomposition strategies in Sec. 4.2. Specifically, we categorize the five federated spatial queries into *radius-known* and *radius-unknown* queries, and elaborate on their decomposition in Sec. 4.3 and Sec. 4.4. We discuss other practical issues in Sec. 4.5.

4.1 Basic Operators

Our acceleration strategy is to *decompose queries into basic operators such that distance-related operations are restricted within silos in plaintext, leaving only secure operations across silos.* The selection of basic operators is explained below.

4.1.1 Operator Selection Principles. We propose two categories of basic operators: *plaintext* and *secure* operators. The plaintext operators perform local queries within each individual silo, while the secure operators securely collect the local query results from different silos as the final output.

- **Plaintext Operators.** They can involve the distance-related operations compulsory in spatial queries, but should be common operations widely supported by diverse spatial databases.
- **Secure Operators.** They should avoid distance-related operations, and efficiently implemented operators are preferable.

Following these principles, we choose two plaintext operators (*plaintext range query*, *plaintext range counting*) and three secure operators (*secure summation*, *secure comparison*, *secure set union*). We define each basic operator and justify our selections below.

4.1.2 Plaintext Operators. We define two plaintext operators: *plaintext range query* and *plaintext range counting*. These operators are performed within each silo F_i. Hence, they can be conducted in plaintext without compromising security.

Definition 5 (Plaintext Range Query/Counting)**.** For a silo F_i, given a query range $\mathcal{R}$, the *plaintext range query* $\mathrm{PRQ}_{F_i}(\mathcal{R})$ returns in plaintext the spatial objects in F_i within $\mathcal{R}$, and the *plaintext range counting* $\mathrm{PRC}_{F_i}(\mathcal{R})$ further returns the number of such objects.

The plaintext operators comply with the principles described in Sec. 4.1.1, because *(i)* the returned results can be securely collected without secure distance operations (see Sec. 4.1.3) and *(ii)* they are supported by almost all spatial databases [43]. These operators are implemented as *plaintext primitives* in Hu-Fu drivers, which we defer to Sec. 5.1. The query range can be a circle, a rectangle or other shapes. For ease of presentation, we focus on a circular range in Sec. 4.2-4.4 and discuss extensions to other shapes in Sec. 4.5.

4.1.3 Secure Operators. Based on the secure multi-party computation techniques [11, 19, 28], we define three secure operators: *secure summation*, *secure comparison*, and *secure set union*. These operators are performed across silos and responsible for secure result collection from local query results returned by plaintext operators.

Definition 6 (Secure Summation)**.** For a federation $F = \{F_1, \cdots, F_n\}$, where each silo F_i holds a number β_i, this operator calculates the sum $\sum_{i=1}^{n} \beta_i$, while avoiding leaking β_i to F_j $(j \neq i)$.

$$\mathrm{SSM}_F(\beta_1, \cdots, \beta_n) = \sum_{i=1}^{n} \beta_i$$

Definition 7 (Secure Comparison)**.** For a federation $F = \{F_1, \cdots, F_n\}$, where each silo F_i holds a number β_i, and a value k, this operator compares $\sum_{i=1}^{n} \beta_i$ with k without leaking $\sum_{i=1}^{n} \beta_i$ or β_i to any F_j $(j \neq i)$.

$$\mathrm{SCP}_F(\beta_1, \cdots, \beta_n, k) = sign\Big(\sum_{i=1}^{n} \beta_i - k\Big)$$

Definition 8 (Secure Set Union)**.** For a federation $F = \{F_1, \cdots, F_n\}$, where each silo F_i holds a set of spatial objects $S_i = \{o_1^i, \cdots, o_{m_i}^i\}$, this operator computes the union of spatial objects from all silos, without leaking the ownership of each $o \in S_i$ to F_j $(j \neq i)$.

$$\mathrm{SSU}_F(S_1, \cdots, S_n) = \bigcup_{i=1}^{n} S_i$$

The secure operators comply with the principles in Sec. 4.1.1 since *(i)* they do not involve distance operations and *(ii)* there are dedicated techniques for efficient implementations (see Sec. 5.2).

4.2 Overall Decomposition Strategies

The principle of query rewriter is to decompose federated spatial queries into as many plaintext operators and as few secure operators as possible such that a large portion of the query can be executed in plaintext without compromising security. At a high level, a federated spatial query is first executed as plaintext operators in each silo, where the results are then securely assembled as the final result. At the minimum, one secure operator is compulsory, and additional secure operators may be necessary if there are extra interactions across silos. Given the basic operators defined in Sec. 4.1, we classify federated spatial queries into two categories and explain their decomposition strategies as follows (see Table 2).

- **Radius-Known Queries.** For a radius-known query, it needs only one secure operator for result collection in the ideal case. This is because our plaintext operators already support plaintext range query and counting. For result collection, a secure set union or summation operator is required. We introduce the decomposition plans for radius-known queries in Sec. 4.3.
- **Radius-Unknown Queries.** For a radius-unknown query, *e.g.*, a federated kNN query, we convert the query into multiple rounds of radius-known queries. There can be cross-silo communication between rounds, and extra secure operators are necessary, which is secure comparison in our case. We adopt binary search to minimize the number of rounds and secure operators involved. We explain the decomposition plans for radius-unknown queries in Sec. 4.4.

4.3 Decomposing Radius-Known Queries

Among the five federated spatial queries, range query, range counting, and distance join belong to radius-known queries.

Decomposition Plan. *Federated range query* can be decomposed into n plaintext range queries with radius r, where each plaintext range query retrieves the local result from each one of n silos. Similarly, *federated range counting* can be decomposed into n plaintext range counting. Observing that a distance join can be viewed as $|R|$ times of range queries with different query points but the same radius, *federated distance join* is decomposed into $|R| \times n$ plaintext range queries with radius r. For result collection across silos, *federated range counting* needs a secure summation to aggregate the counts without revealing the count of any silo. For *federated range query*, it needs a secure set union to assemble the result without revealing the objects' ownership. For *federated distance join*, it also first assembles the results of the plaintext range query in each silo and then executes only one secure set union across silos.

Complexity Analysis. For ease of presentation, we denote the time complexity of plaintext range query/counting as T^q and T^c, respectively. For federated range query/counting, each silo executes a plaintext range query/counting and a secure set union/summation. Thus, their time complexities are $O(T^q + n + |S|)$ and $O(T^c + n^3)$, where $|S|$ is the size of returned set. The communication costs are $O(n + |S|)$ and $O(n^2)$, respectively. For federated distance join, each silo executes $|R|$ plaintext range queries, resulting in a time cost of $O(|R|T^q + n + |S|)$ with $O(n + |S|)$ communication cost. The complexity analysis of secure operators is deferred to Sec. 5.2.

4.4 Decomposing Radius-Unknown Queries

Federated kNN query and kNN join are radius-unknown queries, because there is no specific range in these queries. Thus, their decomposition plan is to first get an appropriate range and then filter the points in the range, as explained in detail below.

Decomposition Plan. Similar to the relation between federated range query and federated distance join in Sec. 4.3, federated kNN join can be viewed as $|R|$ independent federated kNN queries. Hence, we mainly explain how to decompose a federated kNN query.

- **Basic Idea.** Recall from Sec. 4.2, the strategy to decompose radius-unknown queries is to convert them into multiple rounds of radius-known queries. We first derive a radius (denoted by *thres*) via a binary search and then retrieve the spatial objects within this radius. Note that obtaining the exact counting result during the binary search via secure summation may leak extra information of silos. For example, the query user can get the

Algorithm 1: Rewriter of the federated kNN query $\text{kNN}(F, p, k)$ over a data federation $F = \{F_1, \cdots, F_n\}$

1 $[l, u] \leftarrow [0, v_0]$, where v_0 is a predefined upper bound;
2 **while** $u - l \geq \epsilon_0$ **do**
3 $\quad thres \leftarrow (l + u)/2$;
4 $\quad \beta_i \leftarrow$ plaintext range counting $\text{PRC}_{F_i}(\text{circle}(p, thres))$;
5 $\quad sgn \leftarrow$ secure comparison $\text{SCP}_F(\beta_1, \cdots, \beta_n, k)$;
6 $\quad$ **if** $sgn = -1$ **then** $l \leftarrow thres$;
7 $\quad$ **else if** $sgn = 1$ **then** $u \leftarrow thres$;
8 $\quad$ **else break**;
9 $S_i \leftarrow$ plaintext range query $\text{PRQ}_{F_i}(\text{circle}(p, thres))$;
10 query answer $res \leftarrow$ secure set union $\text{SSU}_F(S_1, \cdots, S_n)$;

number of objects within a range, which reveals the federation's data distribution. Hence, we only judge whether the counting result is larger than k and adopt a secure comparison instead. As long as *thres* is between the k^{th} and the $(k+1)^{th}$ nearest distance, the retrieved objects should be the k^{th} nearest neighbors.

- **Algorithm Details.** Alg. 1 illustrates decomposing a federated kNN query. Lines 1-8 derive the radius. We initialize a lower bound ($l = 0$) and upper bound ($u = v_0$) of the radius, where v_0 can be set as the diameter of the area or defined by the user. We then perform a binary search in lines 2-8, where ϵ_0 is the precision lower bound of distance, which can be set as the precision of the locations' coordinates. In each iteration, *thres* is set as $(l + u)/2$ in line 3. For the current radius *thres*, we perform a *plaintext range counting* for each silo in line 4 and a *secure comparison* between the sum of each silo's count (β_i) and the integer k in line 5. Lines 6-8 adjust the boundary of the searching radius. If the total count is smaller than k, the current radius is too short, and we update l to *thres* as the new lower bound (line 6). If the total count is larger than k, it means there are sufficient points within *thres* and we update the upper bound u as *thres* in line 7. The binary search guarantees that *thres* is sufficiently close to the k^{th} nearest distance. In the last round (lines 9-10), a *plaintext range query* $\text{PRQ}_{F_i}(circle(p, thres))$ is performed on each silo and we use a *secure set union* to get the final result.

Complexity Analysis. Alg. 1 takes at most $O(\log \frac{v_0}{\epsilon_0})$ rounds to get the threshold (lines 2-8). In each round, the plaintext range counting (line 4) takes $O(T^c)$ time, and the secure comparison (line 5) takes $O(n)$ time. The adjustment of the binary search boundary (lines 6-8) takes $O(1)$ time. After obtaining the final threshold, the algorithm calls a plaintext range query in $O(T^q)$ time to get local results (line 9) and a secure set union in $O(n+k)$ time to assemble the results. Thus, the total time complexity is $O(T^q + k + (n + T^c) \log \frac{v_0}{\epsilon_0})$. In secure comparison (line 5), each silo communicates with the other $n-1$ silos. Thus, the communication cost for a single round is $O(n^2)$ and there are $O(n^2 \log \frac{v_0}{\epsilon_0})$ rounds in total. The communication of secure set union (line 10) is $O(n + k)$. The time complexity of federated kNN join is similar to federated kNN query, multiplied by a factor $|R|$, *i.e.* $O(|R|T^q + |R|k + |R|(n + T^c) \log \frac{v_0}{\epsilon_0})$.

Example 3. We illustrate the execution of a federated kNN query with query point (4, 4) and $k = 4$ over 3 silos in Fig. 4, and the objects marked with the same color belong to the same silo. The query rewriter decomposes this query into multiple rounds of radius-known queries. In the 1st round, a plaintext range counting with center (4, 4) and radius 4 is sent to each silo and a secure comparison with k is performed across silos. And we get 9 objects, which is greater than k. Hence in the 2nd round, the radius decreases to 2 and resent to silos for plaintext range counting and secure comparison. There are 2 objects, which is smaller than k. Thus in the 3rd round, the radius increases to 3 and the procedure continues, where the range counting result equals to k and the search terminates. Finally, a plaintext range query with center (4, 4) and radius 3 plus a secure set union are performed to get the 4 objects.

4.5 Discussions

We highlight the following discussions on the query rewriter.

Security of Rewriter. We prove the security of our query rewriter based on the composition lemma in [24] (Section 7.3.1). The idea is to show the decomposition plans for radius-known queries and radius-unknown queries will not reveal any extra information other than the final result due to the usage of secure operators. We also present a case study that proves it is hard for a semi-honest adversary to attack Hu-Fu. Please refer to Appendix A of our full paper [15] for the proof and case study due to the page limitation.

Differential Privacy to Accelerate Radius-Unknown Queries. We exploit differential privacy [34] to further accelerate federated kNN query and federated kNN join from two aspects.

- **Tighten Predefined Upper Bound.** We ask each F_i to conduct a local kNN query in plaintext and return the k^{th} object's distance to the query point d_i^k. Since directly returning such value may expose the real distances of silos, we apply the truncated Laplace mechanism [5] on it. That is, let each silo add a positive noise and get the perturbed value $d_i'^k$. We can tighten the upper bound as the smallest distance in all silos, *i.e.* $v_0 = \min_i d_i'^k$, since there are at least k points in this range.
- **Reduce Running Time and Communication Cost in Secure Comparison.** The secure comparison in Alg. 1 compares $\sum_1^n \beta_i$ with k, which incurs at least $O(n^2)$ running time and communication cost. It can be reduced to $O(n)$ when $\sum_1^n \beta_i$ notably differs from k. In this case, each silo can add a Laplacian noise [34] on its local counting result to hide the real counts of each silo, and then aggregate the perturbed results. If the perturbed result is much smaller/larger than k, we directly adjust the threshold.

Beyond Mainstream Spatial Queries. The decomposition plan for radius-known queries applies to federated range query/counting with other query range types (*e.g.*, rectangle). This is because the plaintext range query/counting with arbitrary shapes of query ranges is supported in each silo's underlying spatial data systems (*e.g.*, PostGIS). The query rewriter also supports aggregation queries, *e.g.*, the aggregate attribute on the result of kNN query or range query. Specifically, the aggregation of kNN query can be decomposed the same as the federated kNN query, by only replacing the last secure set union with a secure summation. The range aggregate query can be decomposed similarly to a federated range counting.

5 DRIVERS

This section presents Hu-Fu's drivers, which offer interfaces and implementations on top of silos' spatial databases for the unified and

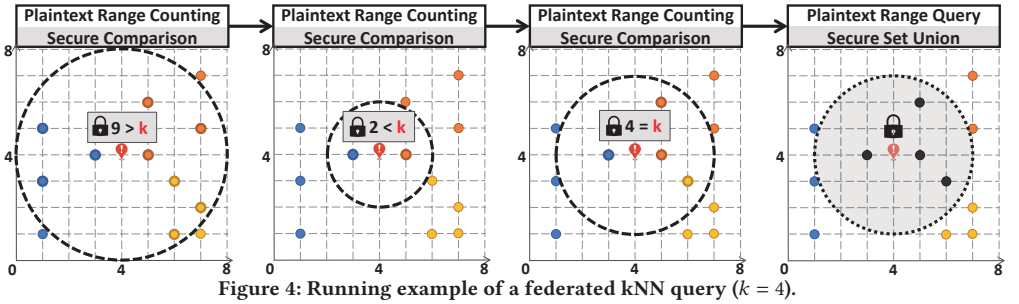


Figure 4: Running example of a federated kNN query ($k = 4$).

Table 2: The number of basic operators in the decomposition plans of federated spatial queries. Radius-known queries only involve one secure operator (secure set union/summation) for secure result collection. Radius-unknown queries are executed in multiple rounds which require extra secure operator (secure comparison) to ensure security. Here, n is the number of silos.

Category	Federated Spatial Query	Number of Plaintext Operator		Number of Secure Operator	
		Range Query	Range Counting	Comparison	Set Union/Summation
Radius-Known	Federated Range Query	n	0	0	1/0
	Federated Range Counting	0	n	0	0/1
	Federated Distance Join	$n\|R\|$ †	0	0	1/0
Radius-Unknown	Federated kNN Query	n	$O(n \log \frac{v_0}{\epsilon_0})$ ‡	$O(\log \frac{v_0}{\epsilon_0})$	1/0
	Federated kNN Join	$n\|R\|$	$O(n\|R\| \log \frac{v_0}{\epsilon_0})$	$O(\|R\| \log \frac{v_0}{\epsilon_0})$	1/0

† $|R|$ is the size of the input dataset R in the federated distance join and federated kNN join.
‡ v_0 and ϵ_0 are user-defined parameters for processing the federated kNN query and federated kNN join.

efficient execution of decomposition plans generated by the query rewriter. A driver, which consists of *plaintext primitives* and *secure primitives*, is deployed on each silo. Upon receiving a decomposition plan, plaintext operators are first executed at each silo with plaintext primitives and then secure operators are performed via the secure primitives for result assembling. As next, we elaborate on plaintext primitives (Sec. 5.1) and secure primitives (Sec. 5.2) to efficiently implement the basic operators defined in the query rewriter.

5.1 Plaintext Primitives

The plaintext primitives implement plaintext range query and plaintext range counting. They are implemented as an *interface* on top of the underlying spatial databases for portability and to harness existing range query and range counting implementations.

Primitive Implementation. The implementation of plaintext primitives is dependent on the underlying spatial databases.

- For databases where range query and range counting are available, *e.g.*, Simba [64] and PostGIS [45], we directly call the corresponding functions for plaintext range query or counting. For example, in PostGIS, a plaintext range counting on silo F_i with the center p and radius r of a circular range can be implemented by calling the SQL below.

```
SELECT COUNT(*) FROM F_i
WHERE ST_DWithin(p, F_i.location, r);
```

- For databases without such queries, drivers provide a default implementation of range query and range counting. For example, GeoMesa [27] only provides an interface of range query. Thus, we implement range counting by first running a range query, and then counting the cardinality of the returned set.

Time Complexity. The time complexity of plaintext primitives depends on the native implementation in spatial databases. For example, plaintext range counting takes $O(\log m)$ time with spatial indices [46], where m is the data size. Yet plaintext range query may need $O(\log m + |S|)$ time, where S is the query result.

Discussions. We make two notes on the plaintext primitives.

- To support the differential privacy based acceleration for federated kNN query (see Sec. 4.5), Hu-Fu drivers provide an optional plaintext kNN query interface. The plaintext kNN is implemented by a function call on spatial databases with native kNN query (*e.g.*, PostGIS [45] and Simba [64]).
- Since the time complexity of plaintext primitives varies, the efficiency of federated spatial queries can be limited by the slowest plaintext primitive if silos are using heterogeneous spatial databases (see Sec. 7.4). Thus, more efficient plaintext range query/counting is out of our scope.

5.2 Secure Primitives

The Secure primitives implement secure summation, comparison, and set union, which are independent of the underlying spatial databases. Recall that secure primitives take the local results from plaintext primitives as inputs. To avoid idle waiting for slow silos and to reuse local results across silos, each silo buffers its local results of plaintext primitives executed on itself. We implement secure primitives on top of such a buffer, as explained next.

Primitive Implementation. Each secure primitive is implemented with a dedicated secure protocol for higher efficiency than the corresponding operation in general-purpose SMC libraries. We present the details of each secure primitive below.

Secure Summation. The implementation of secure summation is based on Ref. [19]. First, all of the n silos first agree on n different public parameters $U = \{u_1, u_2, \cdots, u_n\}$. Then, each silo F_i chooses a random $n-1$ degree polynomial $t_i(x) = (\sum_{k=1}^{n-1} a_{ik}x^k) + v_i$ and calculates n values of the polynomial, $t_i(u_1), \cdots, t_i(u_n)$. Specially, a_{ik} indicates the random coefficient independently generated by silo F_i, and v_i denotes the local counting result of silo F_i. These variables are held by silo F_i only and kept secret from others. Afterward, each silo F_i sends the value of polynomial $t_i(u_j)$ to all other $F_j (i \neq j)$.

When any silo F_j receives all $\{t_i(u_j)|i \neq j\}$ from the other silos, it sums up those values $S(u_j) = \sum_{i=1}^{n} t_i(u_j) = (\sum_{k=1}^{n-1}(u_j^k \sum_{i=1}^{n} a_{ik})) + \sum_{i=1}^{n} v_i$ and sends the summations $S(u_j)$ to the query user. The user can regard $S(u_j)$ as a linear equation $S(u_j) = \sum_{k=1}^{n-1} u_j^k z_k + z_n$, where n unknown variables are $z_k = \sum_{i=1}^{n} a_{ik}$ (for $k = 1, \cdots, n-1$) and $z_n = \sum_{i=1}^{n} v_i$. Moreover, the user knows $\{u_1, u_2, \cdots, u_n\}$ and the values $\{S(u_1), S(u_2), \cdots, S(u_n)\}$. Thus, the user can solve the n unknown variables by Gauss elimination and get the value of $\sum_{i=1}^{n} v_i$ (*i.e.* the unknown variable z_n).

Secure Comparison. The primitive compares a user given constant k with the sum of each silo's local result (*e.g.*, $\{\beta_i\}$) and ensures that either β_i or $\sum_{i=1}^{n} \beta_i$ is confidential to any silos $F_j (j \neq i)$ and the query user. The main idea is calculating $X(\sum_{i=1}^{n} \beta_i - k)$ instead of $\sum_{i=1}^{n} \beta_i - k$, where X is a random and positive real number, because the latter result discloses the value of $\sum_{i=1}^{n} \beta_i$. Accordingly, we reduce our secure comparison into the classic secure multiplication and hence adopt the existing secure multiplication protocol [11] to ensure security. Specifically, the secure multiplication protocol requires two multipliers x and y are both divided into n shares $X = \sum_{i=1}^{n} x_i, Y = \sum_{i=1}^{n} y_i$ and each share is distributed into n silos, *e.g.*, x_i and y_i for silo F_i. This protocol can protect the values of X, Y, x_i, y_i from the attackers in all n silos. In our reduction, Y equals $\sum_{i=1}^{n} \beta_i - k$ and $y_i = \beta_i - \frac{k}{n}$. Since each silo has already known its local result β_i, the user only sends $\frac{k}{n}$ to all silos. After that, each silo randomly generates a positive real number x_i and calculates $XY = (\sum_{i=1}^{n} x_i)(\sum_{i=1}^{n}(\beta_i - \frac{k}{n}))$ by using the secure multiplication protocol (see [11] for more details), then returns XY to the user. Finally, the user derives the final result of our secure comparison by the sign of XY without leaking any sensitive information.

Secure Set Union. We implement this primitive as a random shares based two-phase union method [28]. Specifically, each silo adds its results and some fake records to a global set in the first phase and removes them from the set in the second phase. We use differential privacy to reduce the number of fake records and thus the communication cost. Observing that adding and removing fake records can be done independently, we split the global set into batches to allow parallel execution. Then each silo can add and remove noise data from each batch independently, resulting in a shorter latency.

Complexity Analysis. For secure summation, the time complexity to solve the linear equations is $O(n^3)$, with $O(n^2)$ communication cost [19]. For secure comparison, the time complexity of the secure multiplication is $O(n)$. It also has a communication cost of $O(n^2)$ [11]. The time complexity and communication cost of secure set union are both $O(n + |S|)$, where S is the final global set [28].

6 QUERY INTERFACE

This section presents the query interface of Hu-Fu. For easy usability, the interface offers a unified federation view to users (Sec. 6.1) and supports federated spatial queries in SQL (Sec. 6.2).

6.1 Unified Federation View

Hu-Fu's query interface provides a federation view to the query user, while the detailed information of silos is hidden. This allows the user to send queries without worrying about the silo organization and also protects the data security of individual silos.

We implement the unified federation view by extending the schema manager of Calcite [8], a popular query processing framework. In Calcite's schema manager, each table is independent and indivisible. We add silo as an abstraction layer below the table of schema manager. Thus each table contains multiple silo objects, and each object records the identity information of the corresponding silo. The silo identity information is used when executing secure primitives. Specifically, the query rewriter will attach the identifying information of all silo-level tables in the table of schema manager when distributing secure operators. Each silo only executes the corresponding secure primitives if the attached identity information matches the one locally stored.

6.2 Federated Spatial Queries in SQL

Based on the unified federation view, Hu-Fu query interface supports federated spatial queries in SQL by extending the SQL parser of Calcite. The semantics are almost the same as regular SQL queries. Specifically, we add two keywords: `DWithin` and `kNN`.

For example, a federated range counting on a circular range centered at the point p with radius r can be written in SQL as

```
SELECT COUNT(*) FROM F WHERE DWithin(p, F.location, r);
```

where `DWithin`$(p, F.location, r)$ returns whether the distance between p and an object $o \in F$ is shorter than r. A federated kNN join on a dataset R and federation F with k can be written in SQL as

```
SELECT R.id, F.id FROM R JOIN F
ON kNN(R.location, F.location, k);
```

where `kNN`$(R.location, F.location, k)$ indicates whether a spatial object $o' \in F$ is in the kNN set of query point $o \in R$. Other queries can be written as SQL similarly with these two keywords.

7 EVALUATION

This section presents the evaluations of Hu-Fu. We first introduce the experimental setup (Sec. 7.1), and then present the overall performance (Sec. 7.2), scalability (Sec. 7.3) and results with heterogeneous spatial databases across silos (Sec. 7.4).

7.1 Experimental Setup

Datasets. We conduct experiments on the following two datasets, where each spatial object has a location and a unique ID.

- **Multi-company Spatial Data in Beijing (BJ).** This dataset was collected by 10 companies in Beijing, in June 2019, which has 1, 029, 081 spatial objects in total. The locations of these objects fall into an area from 39.5°N ~ 42.0°N and 115.5°E ~ 117.2°E. We use the dataset to simulate a real-world federation, where each company can be naturally regarded as a silo. We do not alter the distributions of spatial objects across silos, and only vary the silo number n or query-specific parameters (*e.g.*, k for federated kNN query) during the evaluation.
- **OpenStreetMap (OSM).** This is a popular open dataset to evaluate large-scale spatial analytics [16, 43, 64]. We mainly use this dataset in the scalability test, where we sample 10^4-10^9 spatial objects from the Asia dataset in the OpenStreetMap [40]. Specifically, to simulate the spatial overlaps as in the BJ dataset, we assign a random silo ID for each point in the dataset and make each silo have the same number of data points.

Baselines. We compare Hu-Fu with the following baselines.

- **Public.** It directly collects local results from each silo without any secure operation, and serves as the upper bound of query processing efficiency.
- **SMCQL-GIS & SMCQL-GISext.** It adopts the principles of SMCQL [4], a garbled circuit (GC) based solution for relational data, to support spatial queries. We implement SMCQL-GIS and SMCQL-GISext with ObliVM [35], which is also used in SMCQL and only supports two silos. So we only provide the results of SMCQL-GIS and SMCQL-GISext over two silos. And SMCQL-GISext is a variant of SMCQL-GIS without assuming an honest broker, and uses our secure set union to assemble results.
- **Conclave-GIS & Conclave-GISext.** It adopts the principles of Conclave [57], the state-of-the-art secret sharing (SS) based federation solution for relational data, to support spatial queries. Note that we implement Conclave-GIS and Conclave-GISext with a different SS based library, MP-SPDZ [29], rather than Sharemind [11] in the original Conclave. Because Sharemind is devised for only three silos [7] and it is a commercial library. In contrast, MP-SPDZ is a popular open-source library that supports more than three silos based on secret sharing. And Conclave-GISext is a variant of Conclave-GIS without assuming an honest broker, and uses our secure set union to assemble results.

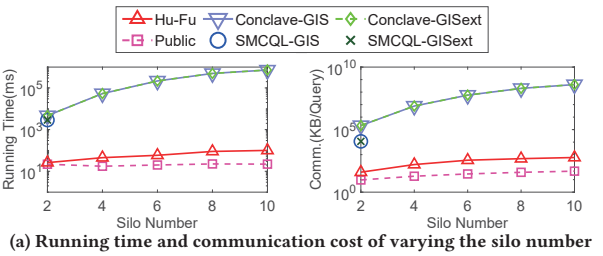
These secure baselines implement federated spatial queries by exploiting similar queries for relational data in SMCQL or Conclave. Our extensions follow the strategy of having plaintext spatial queries within each silo's database and securely computing the final results. Specifically, for *federated range query*, these baselines execute plaintext range query in each silo and collect the partial results by either the honest broker or our secure set union. For *federated range counting*, they execute plaintext range counting and use secure summation to compute the final result. For *federated kNN query*, we regard it as a top-k query with a user-defined function (UDF). For example, each silo runs plaintext kNN query to compute k candidate neighbors along with their distances to the query object. Then, all n silos securely find the k nearest neighbors among nk candidates. For *federated distance join/kNN join*, we refer to their query plans for join queries and regard a federated distance/kNN join as multiple federated range/kNN queries.

Metrics. We assess the query processing efficiency by two metrics.

- **Running time.** It is the time cost from receiving the query from a user to returning the query result to the user.
- **Communication cost.** It is the total network communication among the user and all silos for this query.

All the experimental results are the average of 50 repetitions.

Environment. We run all experiments on a cluster of 11 machines. Each machine has 32 Intel(R) Xeon(R) Gold 5118 2.30GHz processors and 64GB memory with Ubuntu 18.04 LTS. The network bandwidth between machines is up to 10 GB/s. Among the 11 machines, one is as the user and the honest broker for SMCQL-GIS and Conclave-GIS, and the other 10 are data silos. We use PostgreSQL 10.15 with PostGIS extension as the default spatial database for all silos. To show the support of heterogeneous spatial database systems by Hu-Fu, we also use MySQL 5.7 [61], Sqlite3 with SpatiaLite extension [51], GeoMesa 3.0.0 [27], Simba 1.0 [64] and SpatialHadoop 2.4.3 [16] as different silos, as will be explained in Sec. 7.4. They all use spatial indexes (R-Tree in PostGIS, Simba, SpatialHadoop and MySQL, R*-Tree in SpatiaLite, and Z-Curve in GeoMesa) to speed up plaintext primitives by up to 2042× (in Appendix F [15]).



(a) Running time and communication cost of varying the silo number

(b) Running time and communication cost of varying the value of k

Figure 5: Performance of federated kNN query.

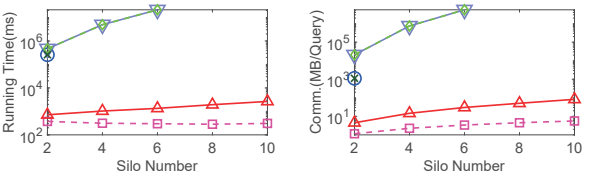
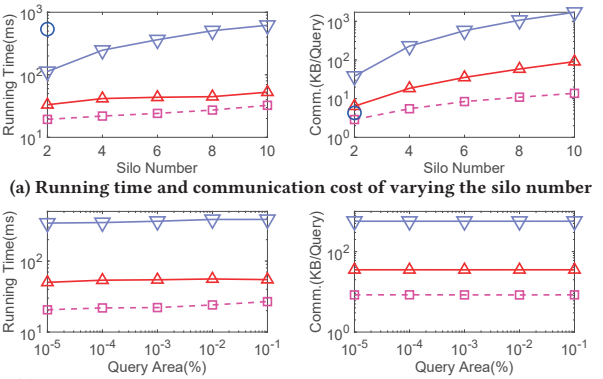


Figure 6: Performance of federated kNN join.



(a) Running time and communication cost of varying the silo number

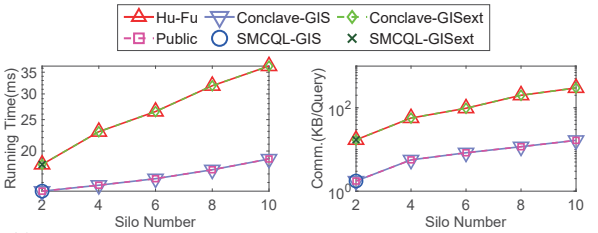
(b) Running time and communication cost of varying the query area

Figure 7: Performance of federated range counting.

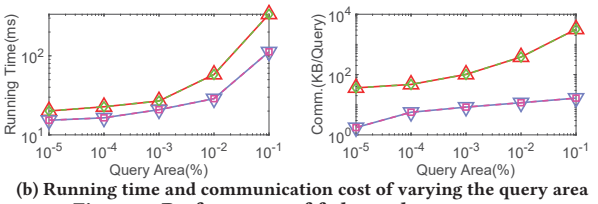
7.2 Results on Real Dataset

This series of experiments compare the efficiency of different methods for all five federated spatial queries on the real dataset BJ. All the query points are randomly sampled from the dataset. We vary the number of silos from 2 to 10, and also test the impact of query-specific parameters. We set k to 16 for federated kNN query and kNN join, and the default query area of federated range query, range counting and distance join as 0.001%, and vary them from 4 to 64 and 0.00001% to 0.1% respectively. The range of these query-specific parameters is aligned with previous study [64]. When evaluating the query-specific parameters, we use 6 silos by default.

7.2.1 Performance of Federated kNN Query. Fig. 5a shows the running time and communication cost of federated kNN query. Hu-Fu is 109.6× to 7,198.8× faster than SMCQL-GIS and Conclave-GIS, and has 2 to 5 orders of magnitude lower communication cost. When the number of silos increases from 2 to 10, the running time and



(a) Running time and communication cost of varying the silo number



(b) Running time and communication cost of varying the query area

Figure 8: Performance of federated range query.

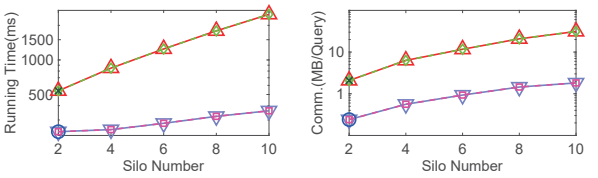


Figure 9: Performance of federated distance join.

communication cost of Hu-Fu only increase by up to 2.9× and 13.9×, while those of Conclave-GIS drastically increase by up to 153.3× and 1,884.3×. Both metrics of Hu-Fu increase because the secure comparison and secure set union used in this query grow linearly with the silo number. Compare with Conclave-GIS and SMCQL-GIS, the running time and communication cost of Conclave-GISext and SMCQL-GISext increase marginally (less than 20 ms and 200 KB, respectively), which shows that our secure set union can efficiently assemble query results without an honest broker.

We also vary k from 4 to 64 and plot the running time and communication cost in Fig. 5b. As k increases from 4 to 64, the running time and communication cost of Hu-Fu only increase by 0.1× and 1.1×, while those of Conclave-GIS increase by 51.3× and 50.7×. The impact of k is less obvious than the silo number on Hu-Fu, because only the secure set union is linearly dependent on k. Again, the efficiency of Conclave-GISext is similar to that of Conclave-GIS. The drastic increase in running time and communication cost of Conclave-GIS and Conclave-GISext is expected because it involves many secure primitives that are time-consuming.

Recall that we apply differential privacy (DP) to accelerate kNN queries (see Sec. 4.5). To prove the gain of the optimization, we list the running time and communication cost with and without DP in Table 3. With DP, the running time is reduced by up to 19.6%, and the communication cost by up to 47.7%. Compared with the improvement, the overhead of injecting the DP noise is very marginal, which takes 2 μs time cost and less than 1 KB communication cost when processing one federated kNN query. Such a notable improvement is because the complexity of DP noise injection is $O(1)$ and the summation only requires for transmission of n integers, while a secure comparison has $O(n)$ time complexity and $O(n^2)$ communication cost.

7.2.2 Performance of Federated kNN Join. Fig. 6 shows the performance of federated kNN join. The results of Conclave-GIS and

Table 3: Ablation of DP optimization for federated kNN query.

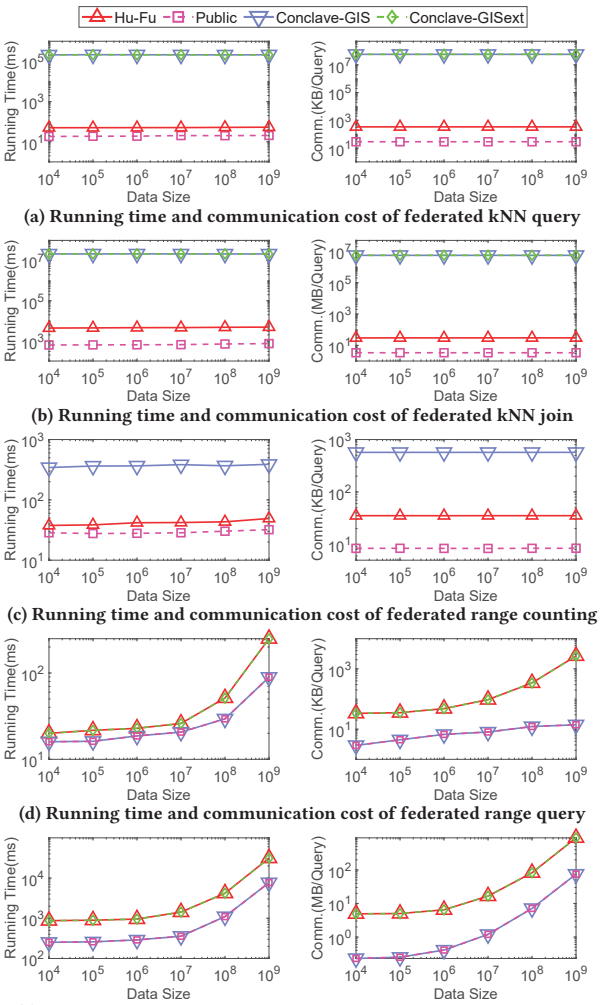
Silo Number		2	4	6	8	10
Running Time (ms)	Hu-Fu	26.1	45.1	58.6	89.6	100.5
	Hu-Fu without DP	26.9	50.3	72.9	107.0	116.9
Comm. (KB)	Hu-Fu	39.4	160.8	357.7	475.1	588.3
	Hu-Fu without DP	58.5	234.8	493.0	784.0	1125.2

Conclave-GISext with over 8 silos are omitted since they incur over 6 hours for a single query. Hu-Fu is still the most efficient, which is up to 360.2×/15,814.2× faster than SMCQL-GIS/Conclave-GIS with 247.8×/185, 151.0× lower communication cost. The running time and communication cost of SMCQL-GISext and Conclave-GISext slightly increase over SMCQL-GIS and Conclave-GIS. The impact of k is similar to federated kNN query (see Appendix B [15]).

7.2.3 Performance of Federated Range Counting. Fig. 7 shows the results of federated range counting. This query only returns the counting result and thus does not need a secure set union to protect data ownership. Hence, we exclude SMCQL-GISext and Conclave-GISext since they only differ from SMCQL-GIS and Conclave-GIS with an extra secure set union, which is unnecessary in this query. Hu-Fu is up to 15.2× faster than SMCQL-GIS with a slightly higher communication cost (within 7 KB). Considering the increasing network bandwidth, the gap in communication cost is acceptable. Compared with Conclave-GIS, Hu-Fu is up to 10.8× faster with 17.9× lower communication cost. The running time and communication cost of Hu-Fu increase by 0.6× and 13.2× respectively when silo number increases to 10, mainly due to the secure summation.

We also demonstrate the impact of the query area on query efficiency in Fig. 7b. As is shown, the running time of all methods is relatively stable. It is expected because secure operations are the bottleneck of running time whereas the larger query area only increases the running time of plaintext operations.

7.2.4 Performance of Federated Range Query. Fig. 8 illustrates the results of federated range query. The efficiency of SMCQL-GIS and Conclave-GIS is the same as Public (*i.e.* the non-secure baseline), because they both rely on an honest broker to securely collect partial answers in each silo without leaking them to any others. Under this assumption, all systems can be reduced to Public, which uses a server (*e.g.*, an honest broker in SMCQL-GIS and a center server in Public) to directly collect local range query result from each silo. For example, Hu-Fu with an honest broker also has the same efficiency as Public (see Appendix D [15]). Under a more general setting without this assumption, Hu-Fu, SMCQL-GISext and Conclave-GISext have the same efficiency because they all use our secure set union for results assembling. The use of secure set union only leads to a marginal increase in running time (within 250 ms) and communication cost (lower than 3.1 MB) over Public. Note that the order of increase in running time and communication cost matches the complexity analysis for the secure set union in Sec. 5.2, which grows linearly with the silo number and the amount of data returned. As shown in Fig. 8b, when the query area expands, all methods have a higher running time and communication cost, due to the increase of the number of spatial objects in the final result.



(a) Running time and communication cost of federated kNN query

(b) Running time and communication cost of federated kNN join

(c) Running time and communication cost of federated range counting

(d) Running time and communication cost of federated range query

(e) Running time and communication cost of federated distance join

Figure 10: Scalability test of federated spatial queries.

7.2.5 Performance of Federated Distance Join. Fig. 9 presents the performance of federated distance join. Note that all the methods treat federated distance join as multiple independent federated range queries, where the total number of these range queries is $|R| = 100$ in this test. Thus, it is reasonable that the ranking of all the methods is similar to that in federated range query (see Fig. 8). The impact of query area is similar to federated range query (see Appendix B [15]).

Takeaways. Overall, Hu-Fu is up to 15,814.2× faster than SMCQL-GIS and Conclave-GIS, with up to 5 orders of magnitude lower communication cost. The efficiency gain of Hu-Fu over the baselines is more notable in federated kNN query, kNN join, and range counting, which is at least 2.4× faster in running time and 4.9× lower in communication cost than Conclave-GIS. SMCQL-GIS and Conclave-GIS are more efficient in federated range query and distance join, because these baselines are reduced to Public and need no secure operation with the honest broker. Note that for federated range query and distance join, Hu-Fu achieves the same efficiency as SMCQL-GISext and Conclave-GISext, the variants of SMCQL-GIS and Conclave-GIS without an honest broker.

7.3 Results on Scalability Test

In this experiment, we scale the total number of spatial objects from 10^4 to 10^9 over OSM dataset to assess the scalability of Hu-Fu. Other parameters are set to the default values as in Sec. 7.2. For example, the number of silos is 6, $k = 16$ for federated kNN query and kNN join, and the query area for federated range query, range counting and distance join is 0.001%. Recall that SMCQL-GIS and SMCQL-GISext only support two silos and are excluded since 6 silos are used in this test. The running time and communication cost on the five spatial queries are shown in Fig. 10.

For a fixed data size, we observe that Hu-Fu is notably more efficient than Conclave-GIS and Conclave-GISext on federated kNN query, kNN join and range counting (see Fig. 10a-10c). For federated range query and distance join, Conclave-GIS behaves the same as Public due to the honest broker, while Hu-Fu achieves the same efficiency as Conclave-GISext, which requires no honest broker.

We are more interested in the efficiency with the increase of data size. We observe that the efficiency of federated kNN query, kNN join and range counting is insensitive to the increase of the data size. This is because the increase of data size mainly affects the time cost of plaintext primitives, which only accounts for a small portion (due to efficient indexes in each silo) in the running time. In contrast, the running time and communication cost of federated range query and distance join notably increase with the increase of the data size because more spatial objects are retrieved in each silo, which leads to a higher cost for both plaintext range query and secure set union.

Takeaways. Hu-Fu trivially scales with data size for federated kNN query, kNN join and range counting because these queries are relatively insensitive to data size. Both metrics of Hu-Fu increase with the data size for federated range query and distance join, yet Hu-Fu is still reasonably efficient for them on large-scale data. For example, in Hu-Fu, a federated range query takes 250 ms running time and 2.6 MB communication cost on the data size of 10^9.

7.4 Results on Heterogeneous Silos

This experiment aims to demonstrate the feasibility of Hu-Fu on heterogeneous spatial databases. Specifically, we use 6 different databases for each silo on the BJ dataset: PostGIS [45], MySQL [61], SpatiaLite [51], Simba [64], GeoMesa [27], and SpatialHadoop [16]. Other parameters are set as the default values as in Sec. 7.2.

Fig. 11 plots the running time breakdown *i.e.* plaintext vs. secure primitives for radius-unknown (*i.e.* federated kNN query) and radius-known (*i.e.* federated range counting) queries (see Appendix C [15] for more results). We make the following observations.

- Given homogeneous underlying spatial databases (PostGIS), our Hu-Fu significantly reduces the running time of secure primitives *e.g.*, 3,935.4× compared with Conclave-GIS for federated kNN query. Such acceleration in secure primitives is the primary contributor to Hu-Fu's gain in running time.
- Heterogeneous underlying spatial databases affect the running time. Specifically, the running time of plaintext primitives is limited by the slowest spatial database, which may increase the overall query processing time. In this experiment, the running time of plaintext primitives notably increases from 4 ms to 579 ms when replacing PostGIS with heterogeneous databases (where

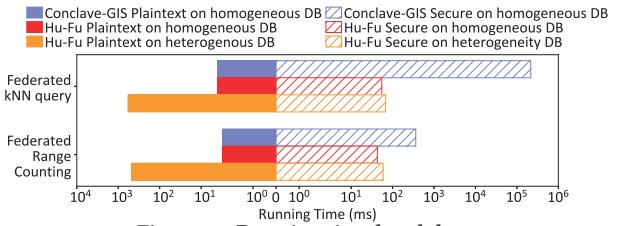


Figure 11: Running time breakdown.

SpatiaLite and MySQL are the slowest), which takes even longer than the secure primitives in Hu-Fu. The running time of secure primitives also marginally increases due to idle waiting for the local results from the slowest silo.

Takeaways. Hu-Fu functions with silos running heterogeneous databases. Although Hu-Fu dramatically speeds up the secure primitives in a federated spatial query, the efficiency of plaintext primitives in each silo's databases may affect the overall running time. Particularly, the time cost of plaintext primitives can be limited by the slowest database in the federation. To unleash the full potential of Hu-Fu, fast spatial databases in each silo are recommended.

8 RELATED WORK

Distributed spatial database systems are popular solutions to query processing on big spatial data. These systems improve query processing via data partition and indexing techniques (*e.g.*, R-tree [46]) in Hadoop (*e.g.*, SpatialHadoop [16] and Hadoop-GIS [2]) or Spark (*e.g.*, Simba [64], GeoSpark [69], and LocationSpark [52]). However, the data partition techniques are inapplicable in a data federation since the entire data is held by the autonomous data silos. Moreover, security is not the major concern in these systems.

Past studies of secure spatial query processing mainly focus on encrypted databases [25], where data is encrypted and stored in a third-party platform (*e.g.*, a cloud platform) to process queries securely. For example, existing work [18, 30, 62, 67] study the secure kNN query on encrypted databases and prior studies [59, 63] focus on securely processing range queries. In these studies, a data owner outsources its data and hence the sensitive data is encrypted before being uploaded to a third party. Intuitively, homomorphic encryption techniques (*e.g.*, Paillier [42] and SEAL [37]) are used to guarantee security. Different from this setting, in a data federation, data silos autonomously manage their own data and hence do not need to encrypt their own data and upload it to a third party.

Rather than the general distributed databases or outsourced databases, our work is more aligned with the problem settings of federated databases and data federation, where the entire dataset is held in multiple autonomous databases. The research on federated databases dates back to 1979 (see surveys [26, 48]). Early efforts focused on finding solutions to access data in autonomous databases [41], while recent studies on federated databases support diverse data types, *e.g.*, on federated graph databases [58]. Note that the autonomous database here means that data can be only managed by its held silo which is different from a self-driving database [32, 33].

Data federation is an emerging concept developed from federated databases. It shares a similar architecture with federated databases. Yet, the *major difference* is that a data federation imposes certain secure requirements during query processing, while a federated database does not. For example, SMCQL [4] is the first secure query processing solution over a data federation and Conclave [57] is the state-of-the-art solution. Wang *et al.* [60] explored join-aggregate queries over a data federation of two silos and Ge *et al.* [23] studied secure functional dependency discovery in a data federation. All these studies adopt SMC techniques to achieve secure query processing for *relational* data with *exact* results.

Other studies investigate *approximate* query processing over a *relational* data federation. For example, Shrinkwrap [5], SAQE [6] and Cryptϵ [14] use differential privacy to trade off between accuracy and efficiency in query processing. In contrast, we focus on *exact* query processing, since accurate results can be crucial for spatial applications like contact tracing [22].

In short, our work is inspired by the emerging trend of secure query processing over a data federation, yet focuses on spatial queries with exact results. Our Hu-Fu significantly improves the efficiency of federated spatial queries over the extensions of SMCQL [4] and Conclave [57], the state-of-the-arts for relational data.

9 CONCLUSION

In this paper, we propose the first system Hu-Fu for efficient and secure spatial queries over a data federation. Existing solutions are inefficient to process such queries due to excessive secure distance operations and the usage of general-purpose secure multi-party computation (SMC) libraries for implementing secure operators. To overcome the inefficiency, we design a novel query rewriter to decompose the spatial queries into as many plaintext operators and as few secure operators as possible. In particular, our secure operators involve no distance operation and have dedicated implementations faster than general-purpose SMC libraries. Moreover, Hu-Fu supports heterogeneous spatial databases (*e.g.*, PostGIS, Simba, GeoMesa, and SpatialHadoop), as well as query input in native SQL. Finally, extensive experiments show that Hu-Fu is up to 4 orders of magnitude faster and takes 5 orders of magnitude lower communication cost than the state-of-the-arts. In the future study, we plan to support more spatial queries and analytics in Hu-Fu, *e.g.*, spatial keyword search.

ACKNOWLEDGMENTS

We are grateful to anonymous reviewers for their constructive comments. Yongxin Tong's work is partially supported by the National Key Research and Development Program of China under Grant No. 2018AAA0101100, the National Science Foundation of China (NSFC) under Grant No. U21A20516, U1811463, 62076017, the State Key Laboratory of Software Development Environment Open Funding No. SKLSDE-2020ZX-07 and WeBank Scholars Program. This research was supported by the Lee Kong Chian Fellowship awarded to Zimu Zhou by Singapore Management University. Yuxiang Zeng and Lei Chen's work is partially supported by National Key Research and Development Program of China Grant No. 2018AAA0101100, the Hong Kong RGC GRF Project 16202218, CRF Project C6030-18G, C1031-18G, C5026-18G, AOE Project AoE/E-603/18, RIF Project R6020-19, Theme-based project TRS T41-603/20R, China NSFC No. 61729201, Guangdong Basic and Applied Basic Research Foundation 2019B151530001, Hong Kong ITC ITF grants ITS/044/18FX and ITS/470/18FX, Microsoft Research Asia Collaborative Research Grant, HKUST-NAVER/LINE AI Lab, Didi-HKUST joint research lab, HKUST-Webank joint research lab grants.

REFERENCES

[1] Acxiom. 2021. https://www.acxiom.com/

[2] Ablimit Aji, Fusheng Wang, Hoang Vo, Rubao Lee, Qiaoling Liu, Xiaodong Zhang, and Joel H. Saltz. 2013. Hadoop-GIS: A High Performance Spatial Data Warehousing System over MapReduce. *PVLDB* 6, 11 (2013), 1009–1020.

[3] AMAP. 2021. https://www.amap.com

[4] Johes Bater, Gregory Elliott, Craig Eggen, Satyender Goel, Abel N. Kho, and Jennie Rogers. 2017. SMCQL: Secure Query Processing for Private Data Networks. *PVLDB* 10, 6 (2017), 673–684.

[5] Johes Bater, Xi He, William Ehrich, Ashwin Machanavajjhala, and Jennie Rogers. 2018. ShrinkWrap: Efficient SQL Query Processing in Differentially Private Data Federations. *PVLDB* 12, 3 (2018), 307–320.

[6] Johes Bater, Yongjoo Park, Xi He, Xiao Wang, and Jennie Rogers. 2020. SAQE: Practical Privacy-Preserving Approximate Query Processing for Data Federations. *PVLDB* 13, 11 (2020), 2691–2705.

[7] Fattaneh Bayatbabolghani and Marina Blanton. 2018. Secure Multi-Party Computation. In *CCS*. 2157–2159.

[8] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. 2018. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. In *SIGMOD*. 221–230.

[9] Paolo Bellavista, Luca Foschini, and Alessio Mora. 2021. Decentralised Learning in Federated Deployment Environments: A System-Level Survey. *ACM Computing Surveys* 54, 1 (2021), 15:1–15:38.

[10] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. 1988. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation (Extended Abstract). In *STOC*. 1–10.

[11] Dan Bogdanov, Sven Laur, and Jan Willemson. 2008. Sharemind: A Framework for Fast Privacy-Preserving Computations. In *ESORICS*. 192–206.

[12] David Chaum, Claude Crépeau, and Ivan Damgård. 1988. Multiparty Unconditionally Secure Protocols (Extended Abstract). In *STOC*. 11–19.

[13] Yong Cheng, Yang Liu, Tianjian Chen, and Qiang Yang. 2020. Federated learning for privacy-preserving AI. *Communications of the ACM* 63, 12 (2020), 33–36.

[14] Amrita Roy Chowdhury, Chenghong Wang, Xi He, Ashwin Machanavajjhala, and Somesh Jha. 2020. Cryptε: Crypto-Assisted Differential Privacy on Untrusted Servers. In *SIGMOD*. 603–619.

[15] Hu-Fu: Efficient and Secure Spatial Queries over Data Federation (Technical Report). 2021. https://github.com/BUAA-BDA/Hu-Fu/blob/dev/Hu-Fu_Technical_Report.pdf

[16] Ahmed Eldawy and Mohamed F. Mokbel. 2015. SpatialHadoop: A MapReduce framework for spatial data. In *ICDE*. 1352–1363.

[17] Ahmed Eldawy and Mohamed F. Mokbel. 2017. The Era of Big Spatial Data. *PVLDB* 10, 12 (2017), 1992–1995.

[18] Yousef Elmehdwi, Bharath K. Samanthula, and Wei Jiang. 2014. Secure k-nearest neighbor query over encrypted data in outsourced environments. In *ICDE*. 664–675.

[19] Fatih Emekçi, Ozgur D. Sahin, Divyakant Agrawal, and Amr El Abbadi. 2007. Privacy preserving decision tree learning over multiple parties. *Data & Knowledge Engineering* 63, 2 (2007), 348–361.

[20] David Evans, Vladimir Kolesnikov, and Mike Rosulek. 2018. A Pragmatic Introduction to Secure Multi-Party Computation. *Foundations and Trends in Privacy and Security* 2, 2-3 (2018), 70–246.

[21] Experian. 2021. https://www.experian.com/

[22] Luca Ferretti, Chris Wymant, Michelle Kendall, Lele Zhao, Anel Nurtay, Lucie Abeler-Dörner, Michael Parker, David Bonsall, and Christophe Fraser. 2020. Quantifying SARS-CoV-2 transmission suggests epidemic control with digital contact tracing. *Science* 368, 6491 (2020), eabb6936.

[23] Chang Ge, Ihab F. Ilyas, and Florian Kerschbaum. 2019. Secure Multi-Party Functional Dependency Discovery. *PVLDB* 13, 2 (2019), 184–196.

[24] Oded Goldreich. 2009. *Foundations of cryptography: volume 2, basic applications*. Cambridge university press.

[25] Hakan Hacigümüs, Balakrishna R. Iyer, Chen Li, and Sharad Mehrotra. 2002. Executing SQL over encrypted data in the database-service-provider model. In *SIGMOD*. 216–227.

[26] Dennis Heimbigner and Dennis McLeod. 1985. A Federated Architecture for Information Management. *ACM Transactions on Information Systems* 3, 3 (1985), 253–278.

[27] James N. Hughes, Andrew Annex, Christopher N. Eichelberger, Anthony Fox, Andrew Hulbert, and Michael Ronquest. 2015. GeoMesa: a distributed architecture for spatio-temporal fusion. In *SPIE*. 94730F.

[28] Pawel Jurczyk and Li Xiong. 2011. Information Sharing across Private Databases: Secure Union Revisited. In *SocialCom/PASSAT*. 996–1003.

[29] Marcel Keller. 2020. MP-SPDZ: A Versatile Framework for Multi-Party Computation. In *CCS*. 1575–1590.

[30] Manish Kesarwani, Akshar Kaul, Prasad Naldurg, Sikhar Patranabis, Gagandeep Singh, Sameep Mehta, and Debdeep Mukhopadhyay. 2018. Efficient Secure k-Nearest Neighbours over Encrypted Data. In *EDBT*. 564–575.

[31] Jinkyu Kim, Heonseok Ha, Byung-Gon Chun, Sungroh Yoon, and Sang K. Cha. 2016. Collaborative analytics for data silos. In *ICDE*. 743–754.

[32] Jan Kossmann, Martin Boissier, Alexander Dubrawski, Fabian Heseding, Caterina Mandel, Udo Pigorsch, Max Schneider, Til Schniese, Mona Sobhani, Petr Tsayun, Katharina Wille, Michael Perscheid, Matthias Uflacker, and Hasso Plattner. 2021. A Cockpit for the Development and Evaluation of Autonomous Database Systems. In *ICDE*. 2685–2688.

[33] Guoliang Li, Xuanhe Zhou, Ji Sun, Xiang Yu, Yue Han, Lianyuan Jin, Wenbo Li, Tianqing Wang, and Shifu Li. 2021. openGauss: An Autonomous Database System. *PVLDB* 14, 12 (2021), 3028–3041.

[34] Ninghui Li, Min Lyu, Dong Su, and Weining Yang. 2016. *Differential Privacy: From Theory to Practice*. Morgan & Claypool Publishers.

[35] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. 2015. ObliVM: A Programming Framework for Secure Computation. In *S&P*. 359–376.

[36] Xiaoyuan Liu, Ni Trieu, Evgenios M. Kornaropoulos, and Dawn Song. 2020. BeeTrace: A Unified Platform for Secure Contact Tracing that Breaks Data Silos. *IEEE Data Engineering Bulletin* 43, 2 (2020), 108–120.

[37] Microsoft SEAL (release 3.6). 2021. https://github.com/Microsoft/SEAL

[38] China Mobile. 2021. https://www.chinamobileltd.com

[39] State of California Department of Justice. 2018. California Consumer Privacy Act (CCPA). https://oag.ca.gov/privacy/ccpa

[40] OpenStreetMap. 2021. https://www.openstreetmap.org

[41] M. Tamer Özsu and Patrick Valduriez. 2020. *Principles of Distributed Database Systems, 4th Edition*. Springer.

[42] Pascal Paillier. 1999. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In *EUROCRYPT*. 223–238.

[43] Varun Pandey, Andreas Kipf, Thomas Neumann, and Alfons Kemper. 2018. How Good Are Modern Spatial Analytics Systems? *PVLDB* 11, 11 (2018), 1661–1673.

[44] European Parliament and The Council of the European Union. 2016. The general data protection regulation (GDPR). https://eugdpr.org

[45] PostGIS. 2021. https://www.postgis.org/

[46] Hanan Samet. 2006. *Foundations of multidimensional and metric data structures*. Academic Press.

[47] Facebook scandal: Who is selling your personal data? 2018. https://www.bbc.com/news/technology-44793247

[48] Amit P. Sheth and James A. Larson. 1990. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys* 22, 3 (1990), 183–236.

[49] Yexuan Shi, Yongxin Tong, Zhiyang Su, Di Jiang, Zimu Zhou, and Wenbin Zhang. 2021. Federated Topic Discovery: A Semantic Consistent Approach. *IEEE Intell. Syst.* 36, 5 (2021), 96–103.

[50] Tianshu Song, Yongxin Tong, and Shuyue Wei. 2019. Profit Allocation for Federated Learning. In *IEEE BigData*. 2577–2586.

[51] SpatiaLite. 2021. http://live.osgeo.org/en/overview/spatialite_overview.html

[52] MingJie Tang, Yongyang Yu, Qutaibah M. Malluhi, Mourad Ouzzani, and Walid G. Aref. 2016. LocationSpark: A Distributed In-Memory Data Management System for Big Spatial Data. *PVLDB* 9, 13 (2016), 1565–1568.

[53] China Telecom. 2021. http://www.chinatelecom-h.com/

[54] Numerous Beijing Taxi Brands to Collectively Connect to Amap's Ride-hailing Platform to Enable Online Operation. 2021. https://aag.cc/newsinfo/517126.html

[55] Communication travel card. 2021. https://xc.caict.ac.cn/

[56] Paul Voigt and Axel Von dem Bussche. 2017. *The EU General Data Protection Regulation (GDPR): A Practical Guide*. Vol. 10. Springer International Publishing.

[57] Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. 2019. Conclave: secure multi-party computation on big data. In *EuroSys*. 3:1–3:18.

[58] Xuan-Son Vu, Addi Ait-Mlouk, Erik Elmroth, and Lili Jiang. 2019. Graph-based Interactive Data Federation System for Heterogeneous Data Retrieval and Analytics. In *WWW*. 3595–3599.

[59] Peng Wang and Chinya V. Ravishankar. 2013. Secure and efficient range queries on outsourced databases using Rp-trees. In *ICDE*. 314–325.

[60] Yilei Wang and Ke Yi. 2021. Secure Yannakakis: Join-Aggregate Queries over Private Data. In *SIGMOD*. 1969–1981.

[61] MySQL (with supports to GIS). 2021. https://www.mysql.com/

[62] Wai Kit Wong, David Wai-Lok Cheung, Ben Kao, and Nikos Mamoulis. 2009. Secure kNN computation on encrypted databases. In *SIGMOD*. 139–152.

[63] Songrui Wu, Qi Li, Guoliang Li, Dong Yuan, Xingliang Yuan, and Cong Wang. 2019. ServeDB: Secure, Verifiable, and Efficient Range Queries on Outsourced Database. In *ICDE*. 626–637.

[64] Dong Xie, Feifei Li, Bin Yao, Gefei Li, Liang Zhou, and Minyi Guo. 2016. Simba: Efficient In-Memory Spatial Analytics. In *SIGMOD*. 1071–1085.

[65] Yi Xu, Yongxin Tong, Yexuan Shi, Qian Tao, Ke Xu, and Wei Li. 2019. An Efficient Insertion Operator in Dynamic Ridesharing Services. In *ICDE*. 1022–1033.

[66] Qiang Yang, Yang Liu, Tianjian Chen, and Yongxin Tong. 2019. Federated Machine Learning: Concept and Applications. *ACM Transactions on Intelligent Systems and Technology* 10, 2 (2019), 12:1–12:19.

[67] Bin Yao, Feifei Li, and Xiaokui Xiao. 2013. Secure nearest neighbor revisited. In *ICDE*. 733–744.

[68] Jieping Ye. 2019. Transportation: A Data Driven Approach. In *SIGKDD*. 3183.

[69] Jia Yu, Jinxuan Wu, and Mohamed Sarwat. 2015. GeoSpark: a cluster computing framework for processing large-scale spatial data. In *SIGSPATIAL*. 70:1–70:4.

[70] Yu Zheng, Xing Xie, and Wei-Ying Ma. 2010. GeoLife: A Collaborative Social Networking Service among User, Location and Trajectory. *IEEE Data Engineering Bulletin* 33, 2 (2010), 32–39.

Sortledton: a Universal, Transactional Graph Data Structure

Per Fuchs
TU Munich
per.fuchs@cs.tum.edu

Domagoj Margan
Imperial College London
d.margan15@imperial.ac.uk

Jana Giceva
TU Munich // MDSI
jana.giceva@in.tum.de

ABSTRACT

Despite the wide adoption of graph processing across many different application domains, there is no underlying data structure that can serve a variety of graph workloads (analytics, traversals, and pattern matching) on dynamic graphs with transactional updates.

In this paper, we present Sortledton, a universal graph data structure that addresses the open problem by being carefully optimizing for the most relevant data access patterns used by graph computation kernels. It can support millions of transactional updates per second, while providing competitive performance (1.22x on average) for the most common graph workloads to the best-known baseline for static graphs – CSR. With this, we improve the ingestion throughput over state-of-the-art dynamic graph data structures, while supporting a wider range of graph computations under transactional guarantees, with a much simpler design and significantly smaller memory footprint (2.1x that of CSR).

PVLDB Reference Format:
Per Fuchs, Domagoj Margan, and Jana Giceva. Sortledton: a Universal, Transactional Graph Data Structure. PVLDB, 15(6): 1173 - 1186, 2022.
doi:10.14778/3514061.3514065

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/PerFuchs/gfe_driver.

1 INTRODUCTION

Graph processing on dynamic datasets is an increasingly important problem for many application domains, from recommender systems to fraud and threat detections [25, 46, 50, 51]. Today many scenarios need to perform a wide range of graph computations – analytics, graph pattern matching (GPM), and traversals – on diverse datasets, which can be highly dynamic and entail millions of edge insertions per second [50]. For example, Alibaba uses a combination of graph analytics and interactive graph traversals for fraud detection [17], while the Twitter recommendation service is based on GPM and traversals [25, 51]. Both need to perform the above-mentioned analysis while ingesting many updates per second [17, 51].

Building a system that can efficiently process such a diverse set of graph algorithms over a dynamic graph dataset is still an open problem. This is largely because such a system needs an underlying data structure that can absorb a high rate of transactional updates, while efficiently processing the wide range of heterogeneous graph workloads. Designing such a universal data structure is a non-trivial challenge, that we discuss and address in this paper.

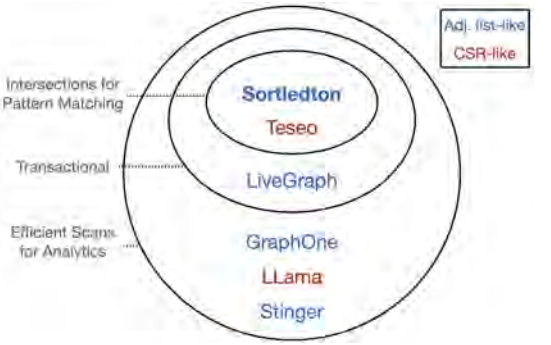


Figure 1: Supporting scans, transactional updates and intersections for a universal graph data structure is challenging.

Figure 1 depicts the related work landscape in the context of supporting such requirements. Most of the existing related work has not succeeded at supporting all of them. Let's focus on the data structure that stores the neighborhood of vertices. First, to achieve a competitive performance on graph analytical workloads (*e.g.*, page rank) or traversals (*e.g.*, single-source shortest path) the data structure needs to support fast scans [57]. Second, to efficiently support GPM (*e.g.*, triangle counting), the data needs to be sorted to enable fast intersections [1, 22, 41]. Third, to support the ingestion of high update rates, the data structure needs to be dynamically adjustable. Finally, to ensure correct results for the concurrently executing analytical queries in the presence of updates [39], the data structure needs to support versioning (*e.g.*, MVCC [43]). An avid reader will notice that satisfying any 2 out of the 3 requirements is relatively simple but the combination of all three is challenging.

As a result, most prior work has focused on static graphs [18, 22, 29, 52] or foregoes support for GPM when operating on dynamic graphs [19, 30, 36, 57]. Furthermore, most dynamic graph data structures do not provide transactional guarantees. Hence, many of the graph algorithms that have been developed for static graphs cannot be run concurrently with updates [19, 30, 36]. Livegraph was the first system to propose a multi-versioned, transactional data structure, by storing two timestamps per edge [57]. However, this triples the memory consumption and hurts the scan performance. To the best of our knowledge, only Teseo [34] provides concurrent support for all three requirements, depicted in Figure 1. However, Teseo imposes a significantly more complex design than us (*CSR-like*), without demonstrating advantages in performance.

We first present a systematic analysis of memory access patterns to support optimal performance, *e.g.*, *sequential vertex access* or *sequential neighborhood access*. Utilizing a framework composed of different basic data structures, we isolate and quantify the effects of access patterns.

Proceedings of the VLDB Endowment, Vol. 15, No. 6 ISSN 2150-8097.
doi:10.14778/3514061.3514065

Based on our insights, we propose Sortledton – an *adjacency list-like* universal data structure that can ingest millions of transactional updates per second while supporting the high performance execution of *heterogeneous* graph workloads. Sortledton uses a sorted set data structure to store neighborhoods, which can be scanned as fast as a contiguous region of memory, while supporting fast intersections for GPM computations and up to 5 million transactional edge updates. We show that Sortledton can outperform all competitors for update throughput. At the same time, we are either faster or on par even with the highly specialized systems for all graph workloads on dynamic datasets and on average only 1.2x slower than running the computations on a static graph stored in CSR [49] format.

2 BACKGROUND AND MOTIVATION

To better motivate the problem, we start by detailing what we mean by heterogeneous graph workloads. Three graph workloads categories exist in the literature: graph analytics, graph pattern matching (GPM), and graph traversals [8]. Notable example algorithms for each category are PageRank, triangle count, and single-source-shortest path. The survey by Sahu et al. finds that all three workload categories are frequent use-cases in graph databases computations [50]. For example, Alibaba uses a combination of graph analytics (finding big bicliques) and interactive graph traversals for fraud detection [17] and Twitter recommends tweets based on GPM as well as traversals [25, 51]. Both companies describe these workloads as highly dynamic [17, 51]. Despite the increasing necessity for efficient support of diverse dynamic graph workloads, most graph processing systems target only a single workload category or static graph computations [18, 22, 29, 52, 57].

2.1 Understanding the Problem

There are two key challenges when designing a universal graph data structure for all aforementioned workload categories on dynamic graphs. The first one is to support a wide range of operations: (1) all workloads require fast scanning of neighborhoods, (2) high throughput of new edges requires fast insertions, and (3) GPM needs intersection [12, 22, 41]. It is relatively easy to achieve a combination of any two of these operations. Scans and inserts can be supported by a vector – with an amortized *push_back* operation. However, intersections are slow because they run in $O(N \times M)$, with N and M being the cardinalities of the participating vectors. Scans and intersections can be supported by a sorted array but this is a static data structure and individual insertions are very slow. Fast intersections and inserts can be supported with a hash set. However, hash sets have empty slots which require the evaluation of a predicate for each scanned element. Hence, supporting all three operations requires a trade-off and we address this with a systematic study in Section 3 and a data structure design in Section 4.

The second challenge is to run updates concurrently alongside computations while maintaining the correct semantics [39, 54]. Most graph algorithms are developed for static graphs. Therefore, they require a static view of the graphs. In dynamic environments, this can be provided by concurrency control systems. In particular, multi-version concurrency control allows us to write to the newest version of the graph and read older static versions. We address this by first identifying the specific transactional requirements in Section 3.5 before describing a graph-optimized concurrency control system in Section 5.

2.2 Memory Access Patterns

Graph workloads are known to be memory access bound [7, 20]. Hence, optimizing for their memory access patterns is most important. We identify four common memory access patterns:

(1) sequential access to the neighborhoods of all vertices
(2) sequential access to the edges within a neighborhood
(3) random access to algorithm-specific properties, *e.g.*, scores for PageRank or distances for BFS
(4) random access to the neighborhoods of all vertices

The PageRank (PR) algorithm in Listing 1 exhibits all access patterns except the second. It accesses all neighborhoods and edges sequentially in the order of the vertices (line 3) and reads the *contributions* array at random locations (line 4). The *random vertex access pattern* typically arises within traversal algorithms.

```
Data: contrib: V-sized array, contributions per vertex this
      round, scores: V-sized array, scores next round
1 for v ∈ V do
2 |   incoming ← 0
3 |   for e ∈ v.neighbors do
4 |   |   incoming ← incoming + contrib[e]
5 |   scores[v] ← incoming
```

Algorithm 1: An example for *sequential vertex access*: the main loop of a PageRank algorithm.

2.3 Graphalytics Benchmark

To quantify the effects of optimizing for different memory access patterns, we use the kernels specified by the LDBC Graphalytics Benchmark [11]. These cover all three graph workload categories and all four access patterns. The benchmark includes 5 kernels: weakly connected component (WCC), PageRank (PR), community detection via label propagation (CDLP), breadth-first search (BFS), weighted single-source shortest path (SSSP), and local clustering coefficient (LCC). The first three are examples of analytical algorithms. The next two are graph traversals and the last one is dominated by triangle counting – a typical GPM algorithm. All algorithms exhibit the *sequential neighborhood access* and *algorithm-specific property access patterns*. The analytical algorithms show *sequential vertex access*, while SSSP and LCC access the neighborhoods in random order. The direction-optimized BFS exhibits both vertex access patterns, but is dominated by *sequential vertex access*.

For WCC, PR, BFS, and SSSP, we used the reference implementation of the *Graph Algorithm Platform Benchmark Suite* (GAP BS) [6]. LCC and CDLP are implemented as in Teseo [34].

The Graphalytics Benchmark also provides the graph dataset (Table 1). The Graph500-x datasets are synthetic power-law graphs. The scale factor x describes the number of edges and vertices in the graph; each increment doubles the number of vertices and edges. Uniform-x datasets are like Graph500-x but they have a uniform degree distribution. *dota-league*, *com-friendster*, *yahoo-songs*, and *edit-wiki* are real-world graphs. The latter two are from the

Table 1: Graph datasets with number of edges and vertices, average degree and size in memory when stored as an undirected graph with 8 Bytes per vertex and 16 Bytes per weighted edge.

Graph	#V	#E	$\overline{D}$	Size [GB]
dota-league	61K	51K	836	1.6
graph500-22, uniform-22	2M	64M	26	2.0
yahoo-songs	1.6M	256M	315	7.6
edit-wiki	51M	255M	22	7.6
graph500-24, uniform-24	9M	260M	29	8.3
graph500-26, uniform-26	33M	1B	33	33.9
com-friendster	29M	2B	72	67.0

Table 2: Operations for a universal graph data structure.

Operation	Complexity	Required for
Basic		
get_neighbors	$O(1)$	all workloads
scan_neighbors	$O(D)$	all workloads
insert_edge	$O(\log D)$	all dynamic workloads
insert_vertex	$O(\log V)$	all dynamic workloads
delete_edge	$O(\log D)$	some dynamic workloads
delete_vertex	$O(D)$[1]	some dynamic workloads
Set functionality		
find_edge	$O(\log D)$	updates, consistency checks
intersect_neighbors	$O(D)$	GPM

KONECT project [31] and are bipartite networks with edge creation timestamps.

3 REQUIREMENTS AND DESIGN GOALS

Now that we understand the key challenges for building a universal graph data structure for dynamic graphs, we discuss our systematic approach to designing one. To address the heterogeneity challenge of Section 2.1, we begin by outlining the requirements for a graph data structure in general, *i.e.*, the necessary operations. They are listed in Table 2. We categorize them into basic and set functionality. The first category is supported by most former graph data structures [19, 30, 36, 57]. However, the second category is not captured by them as they store neighborhoods in list data structures. Hence, their *intersect_neighbors* operation has the complexity of $O(N \times M)$ with N and M being the size of the participating neighborhoods. Similarly, their *find_edge* operation completes in $O(D)$ (D defined as the average degree of the graph). Efficient support of these operations is critical for GPM and dynamic workloads that update or delete edges. Hence, a universal graph data structure should store neighborhoods in set data structures. In the next subsections, we show how the memory access patterns from Section 2.2 influence the design of graph data structures by running multiple microbenchmarks.

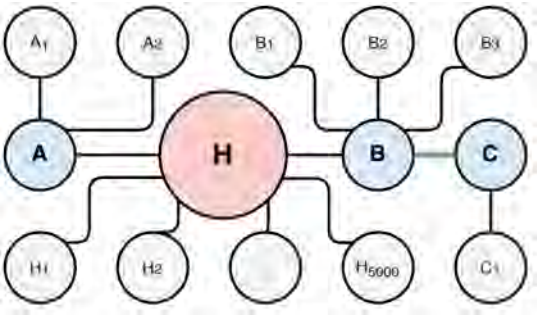


Figure 2: Example graph with hub vertex, H, and vertices (A, B, C) with their neighbors.

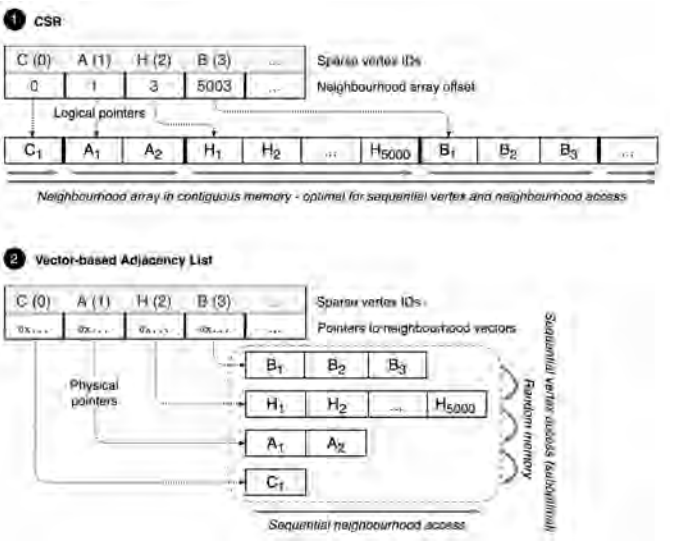


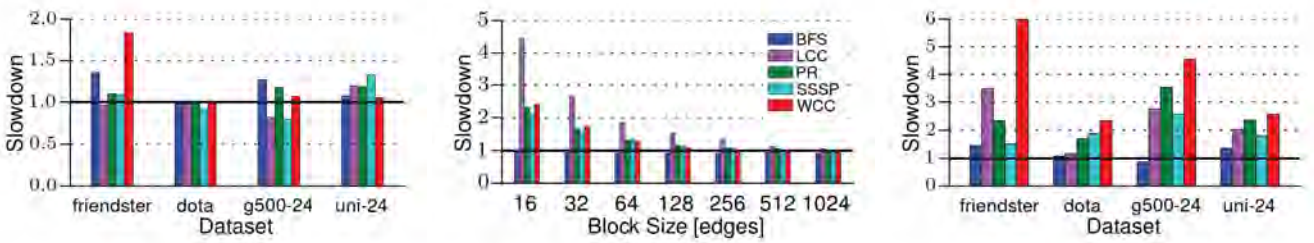
Figure 3: Classical graph data structure designs ① CSR, ② Vector-based adjacency list for example graph from Figure 2.

3.1 Sequential Vertex Access

The first memory access pattern is (1) *sequential vertex access*, *e.g.*, the outer loop of PR (see Listing 1). Ideally, one should store the neighborhoods of all vertices contiguously in memory. The CSR data structure, depicted in Figure 3, is specifically optimized for this access pattern. It is a static structure and it stores all neighbors in a large array in the order of the vertices they belong to.

We analyze the effects of such memory layout optimizations with a simple microbenchmark. We compare the runtimes of the algorithms from the Graphalytics benchmark (c.f. Section 2.3) executed on CSR and a simple sorted vector-based adjacency list, as presented in Figure 3. Such an adjacency list implementation stores the neighborhoods in random memory places with no relationship to each other, thereby representing the other end of the spectrum. Even when using the adjacency list, one can add software prefetching instructions to optimize for the predictable vertex access pattern.

[1]The *delete_vertex* operations cannot be supported in O(log V) due to the fact that all edges which reference this vertex need to be deleted.



(a) ***sequential vertex access*****: vector adjacency list vs CSR.**

(b) ***sequential adjacency access*****: varying edge block sizes vs vectors.**

(c) ***algorithmic-specific properties*****: Teseo sparse vs dense domain.**

Figure 4: Effects of optimizing for different access patterns (cf. Section 2.2.)

We do so only for the BFS algorithm by adding a single prefetch instruction to fetch the neighborhood three vertices ahead.

The normalized runtimes are shown in Figure 4a, where a ratio above 1 means the CSR is faster. We observe that even though PR and WCC heavily rely on sequential vertex access, an adjacency list mostly stays within 40% of the runtime of a CSR. Only WCC on *com-friendster* has a slowdown of over 1.4. LCC and SSSP show speedups of 20% for *graph500-24*. They have a dominant *random vertex access* pattern and we suspect that CSR results in a higher false sharing of the cache lines.

We conclude that while optimizing for the *sequential vertex access* is beneficial, it is not strictly necessary for good analytical performance because optimizing for *sequential vertex access* only addresses a small fraction of all memory accesses, *i.e.*, the first memory access to a new neighborhood. These are in the order of $|V|$ while patterns (3) and (4) can occur in the order of $|E|$ accesses, where E can be at least 10x larger than V [31, 35].

3.2 Sequential Neighbourhoods Access

Next, we analyze the effects of optimizing for the *sequential neighborhood access pattern*. Ideally, one would use a contiguous memory region for each neighborhood, as done by Livegraph [57]. However, no dynamic set data structure with such properties supports intersections. Therefore, we check how well the access pattern can be supported by sorted sets that maintain blocks of elements (*e.g.*, B+ trees [13] or unrolled skip lists [45]).

To do that, we compare the runtimes of the Graphalytics algorithms on: (1) vector-based adjacency list where neighborhoods are completely continuous to (2) the adjacency sets in sorted blocks that are linked together via pointers. We vary the block size. Intuitively, a larger block size would lead to lower run times.

The normalized runtimes are shown in Figure 4b. We note that all algorithms show nearly equal performance on both data structures when (2) uses a block size larger than or equal to 256 edges. We conclude that optimal *sequential neighborhood access* can be supported by set data structures with at least 256 edges per block.

3.3 Random Access to Algorithm Properties

The third memory access pattern is reading the algorithm-specific properties. Thus, it is not influenced by the memory layout of the graph data structure itself. However, we can influence the data structure that holds algorithmic-specific properties by choosing the domain of the vertex identifiers. We hypothesize that it is the most important access pattern because it happens in the innermost loop of the computation and is random, *e.g.*, PR line 4 of Listing 1.

To explore the effects, we test two options for the vertex identifier domain: dense and sparse. Most data structures store the *dense* domain ($[0, \ldots |V|]$) [19, 30, 36, 57], and many graph algorithms are implemented assuming this domain, *i.e.* they use arrays to store algorithmic-specific properties. However, storing a dense domain complicates the deletion of vertices and we cannot assume that the vertex identifiers provided by the user to a graph database are dense [11, 21]. Therefore, we explore storing the *sparse* domain.

Since, our framework for microbenchmarks does not support *sparse* vertex identifier domains, we evaluate the effects of a *sparse* domain vs *dense* domain using Teseo [34]. We show normalized runtimes in Figure 4c. Running on a *dense domain* is very beneficial for most algorithms, leading up to 6x performance improvements. In the case of Teseo, the main overhead is from using a hash map to translate edges from a sparse to a dense domain.

An alternative would be to rewrite all graph algorithms to use concurrent hash maps to store algorithmic-specific properties, *e.g.*, the *contrib* and *scores* array in Listing 1, so, they can run on the sparse domain directly. This, however, would incur similar overheads. Furthermore, it complicates the parallelization of the algorithm due to using concurrent hash maps instead of arrays.

In conclusion, any graph data structure for analytics needs to store a dense vertex identifier domain. In the dynamic setting with user-provided vertex IDs this requires translating a sparse domain into a dense domain when inserting new vertices.

3.4 Random Vertex Access

Finally, the *random vertex access pattern* happens when neighborhoods of vertices are accessed in an unpredictable order. This is typical for graph traversals, *e.g.*, SSSP. Ideally, one aims to minimize the latency for retrieving the neighborhood of a vertex. Given the need for dense vertex identifiers, we can use the IDs as offsets in a vector to store the mapping, as done in many existing data structures [19, 30, 57]. This minimizes lookup latency compared to other mapping data structures like trees and hash sets because vectors need exactly one memory access per lookup. We measure that using a `std::sorted_map` or a robin hood hashmap[2] instead

[2] https://github.com/martinus/robin-hood-hashing

of a vector for the mapping, resulting in slowdowns between 1.1x and 3x, depending on the algorithm and graph.

In conclusion, we establish that a universal dynamic data structure for heterogeneous workloads needs to have:

(1) a set data structure for neighborhoods to run intersections
(2) a neighborhood data structure keeping large blocks of edges for *sequential neighborhood access* (3.2)
(3) ability to expose a sparse and a dense vertex domain to the user (3.3)
(4) a low-latency index for *random vertex access* (3.4)

Furthermore, we find that optimizing for *sequential vertex access* can be beneficial, but is not as critical as for other access patterns.

3.5 Transaction on Graphs

The second challenge we address is the support for running updates and analytics concurrently and in isolation using MVCC to reuse static analytical and GPM solutions in dynamic settings. For this, we identify the transactional requirements of graph workloads.

Concurrency Control: Graph workloads are read-heavy and common read-write transactions are very simple [2, 4, 6, 11, 25, 46, 51]. While queries can take up to tens of seconds for analytics or multiple minutes for GPM, the majority of write transactions add a single edge with corresponding vertices and properties, which are very fast operations. Furthermore, many of the read-write requests are actually writes with a priori known write set [4, 21].

The design of the upcoming SQL standard extension for graph transactions (GQL and SQL/PGQ [23, 28]) reveals further insights into the type of graph transactions that need to be supported. While SQL/PGQ is read-only, GQL supports writes as described earlier as well as *graph construction* [3]. The latter involves complex read-write transactions that can touch large parts of the graph. We summarize that graph transactions fall in the following categories with a descending frequency of occurrence:

(1) long-running and complex read-only queries
(2) short and simple writes with known write set
(3) complex read-write requests with arbitrary large, unpredictable read- and write sets

Hence, an ideal graph concurrency control system should:

(1) decouple read-only queries from write requests
(2) support high throughput writes with a known write set
(3) provide support for read-write transactions with large read/write sets

In this paper, we focus on the first two points. Small read/write sets can be supported by HTAP protocols [43] but large read/write sets combined with highly frequent updates require novel concurrency control mechanisms and are out of scope for this paper.

Version Storage: All entities are small (*e.g.*, an edge takes only 4 or 8 bytes). Thus, versioned storage should induce no overhead for unversioned records and little overhead for versioned edges. Second, vertices and edges have only two states: present or not. Multiple versions can only occur if vertices/edges are inserted, updated, or deleted multiple times before a version can be garbage collected which is rare. Hence, an efficient system should optimize for the case of one and two versions.

Consistency: Most graph systems assume the following consistency guarantees:

- *no dangling edge*: $\forall(a,b) \in E \Rightarrow a \in V \land b \in V$
- *no duplicate edge*
- *reverse edge exists* (for undirected graphs): $\forall(a,b) \in E \rightarrow \exists(b,a) \in E$

Enforcing these rules requires an implicit read set in write-only transactions (*e.g.*, an edge insertion also reads the edge and its vertices to ensure non-existence of the edge and existence of the vertices). These read sets can be determined from the write sets. Concurrency control protocols used in relational databases are a natural way to enforce that these consistency guarantees hold atomically. Hence, transaction handling is also relevant for graph processing and graph streaming systems. The outlined requirements are addressed in Section 5.

4 DATA STRUCTURE DESIGN

Our design implements the operations from Table 2 and is optimized for *random vertex access*, *sequential neighborhood access*, and *algorithm-specific property access*. Supporting only these three access patterns allows a simple design because neighborhoods can be independent data structures.

Finally, our data structure has no hidden costs like amortized operations or background threads, runs analytics without the need for any precomputation, and ingests updates into read-optimized segments directly.

4.1 High-Level Design

There are two high-level designs for graph data structures. We name them *adjacency list-like* and *CSR-like* because their main characteristics stem from these classical data structures (Figure 3). The *adjacency list-like* design has one *adjacency index* and an *adjacency list* for each neighborhood. The neighborhoods are sets `Set<ID>` of destinations, and the index is a map `Map<ID, Set<ID>>`. The *CSR-like* design stores all neighborhoods in one data structure and maintains an index of offsets for it. The formalization of the index is `Map<ID, offset>`. A strawman of the neighborhood data structure is a set of edges: `Set<pair<ID, ID>>`. However, this is neither performant for computation nor space-efficient because it replicates the source of the edges many times. Ideally, we need a set that stores sources and destinations clustered by source.

We compare both designs in terms of the memory access patterns from Section 2.2. Both optimize for *random vertex access* with their indices and neither influences the *algorithm-specific access*. Furthermore, the *CSR-like* design optimizes for both *sequential vertex access* and *sequential neighborhood access*, while the *adjacency list-like* design only optimizes for *sequential neighborhood access*.

Given our insight that optimizing for *sequential vertex access* is less beneficial than for any other pattern (Figure 4), we choose the *adjacency list-like* design. This has three advantages.

First, an *adjacency list-like* data structure is embarrassingly parallel at the granularity of vertices because its neighborhoods are independent. Parallelization at this granularity is successfully utilised in the vertex-centric computation model and many algorithms [6, 37]. Second, the maintenance of the index is simple and cheap because it is independent of changes to the neighborhoods. This is opposed to

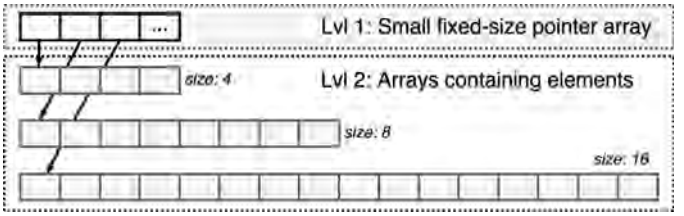


Figure 5: Two-level vector.

the expensive maintenance in the *CSR-like* design where one edge insertion leads to multiple index updates. This requires solutions that impede *random vertex access*, *e.g.* lazy or amortized index updates [34, 36]. Finally, the *adjacency list-like* design allows reusing well-studied map and set data structures from prior research [27]. The *CSR-like* design requires a novel data structure that incorporates the factorization of edges into existing set designs [34]. The key insight is to decompose the problem of building a graph data structure into choosing a map and set type, and parallelizing them. Next, we pick a suitable map and set candidate.

4.2 Data Structure

The adjacency index maps vertex IDs to vertex records. A record contains multiple fields: a pointer to the neighborhood, its size and a read-write latch for parallelization.

As described in Section 3.4, to minimize the latency of a map lookup, we use a vector. To concurrently resize the vector without locking it for updates, we use two levels (Figure 5). The first level is small and has a fixed size. It holds pointers to the second-level segments that contain the vector's elements. When resizing the vector, we allocate exponentially growing second-level segments and add a corresponding pointer in the first level concurrently [14].

The adjacency sets store the neighborhood of each vertex. For a universal graph data structure, they should support intersections and *sequential neighborhood access* (Section 3) - sorted sets that store blocks of edges are well-suited. Typical implementations of such sets are B+ trees [13] and unrolled skip lists [45]. We choose the second because it does not need global rebalancing [26]. In contrast to the original unrolled skip list, we keep edges within blocks sorted. We show this structure in Figure 6 ❸. The elements of the unrolled skip list are blocks of edges combined with the header containing: the number of edges, the highest destination within the block, and pointers for each level of the skip list.

The data structure supports standard set operations by combining ordinary skip list algorithms to find the correct block and then a binary search within the block to find the correct position for reading or writing. Blocks split into two when they fill up, and merge into one when they are less than half full. Therefore, the fill ratio of our block is between 50% and 100%. Both insertions and deletions move at most *block size* elements. Hence, the operations complete in $O(\max(\log D, blocksize))$.

With such properties, an unrolled skip list is a good choice for hub vertices. However, for vertices with neighborhoods smaller than the *block size*, we use headerless, power of two-sized vectors (Figure 6 ❷). This is space-efficient and follows the power-law distribution [57]. Insertions and deletions respect the sorted order and complete in $O(blocksize)$.

The *block size* influences the performance of graph computations (cf. Section 3.2) and edge insertion throughput which we analyze in Figure 7 when loading *Graph500-24* edge-by-edge (meps stands for million edges per second). A *block size* of 128 leads to the highest throughput. With smaller blocks, insertions suffer from random memory access to find the correct block. For larger blocks, insertions need to shift a larger number of edges within the blocks. We expect similar results for other power-law graphs. For uniform graphs, block sizes above the average degree show no influence on performance because all neighborhoods are kept in single blocks.

Vertex identifier translation Analytical workloads require a dense vertex identifier domain but in dynamic settings users usually provide identifiers from sparse domains (Section 3.3). Therefore, we provide a simple binary translation between the domains, and an interface to access both. Performance critical computations should use the dense domain and translate the inputs and outputs as detailed in Section 6. Figure 6 ❶ shows these translations as logical-to-physical and physical-to-logical indices (*lp-index*/*pl-index*). To store the translation, we use a concurrent hash set [47] from sparse to dense, and a concurrent two-level vector from dense to sparse domains.

Edge Properties Edge properties are common in graph analytics, GPM, and traversals. Their storage should be governed by their access patterns as detailed by Gupta et al. [24]. They are usually accessed during scans and should follow the same order as the edges. However, as many workloads do not access them, columnar storage is preferred [24]. Hence, we store them at the end of each edge block, in the same order as the edges (Figure 6).

4.3 Parallelization

We have one read-write latch per vertex to allow multi-threaded access (see Figure 6 ❶). Before executing any operation on the vertex or its edges, the latch needs to be locked. This locking mechanism is simple. It scales because the number of latches in the system grows with the number of vertices. It is dead-lock-free because most operations require only a single latch, and we guarantee a global locking order for intersections and multiple open scans.

Our locking model can lead to scalability bottlenecks when processing hub vertices. To overcome this, one can parallelize the unrolled skip list with one latch per block. That way, we can implement all operations such that at most one block per skip list level needs to be locked at any time [45]. Additionally, we propose optimistic latches [9] for all read operations, but scans. However, we choose the simple locking model because it has better scalability than all competitors (Section 7.2), and leave the concurrent skip list implementation for future work.

5 CONCURRENCY CONTROL

Now, we describe the design of our graph-optimized transaction support that addresses the second challenge of running updates concurrently with computations (Section 3). In relational systems, this is achieved by MVCC protocols [16, 43, 48]. However, in graph transactions, the writes are simpler, and the versioned items (edges and vertices) are at most 8 bytes. Therefore, we adapt prior work from relational systems [43, 48] to these new characteristics with the goal to minimize memory usage and computational overhead. For

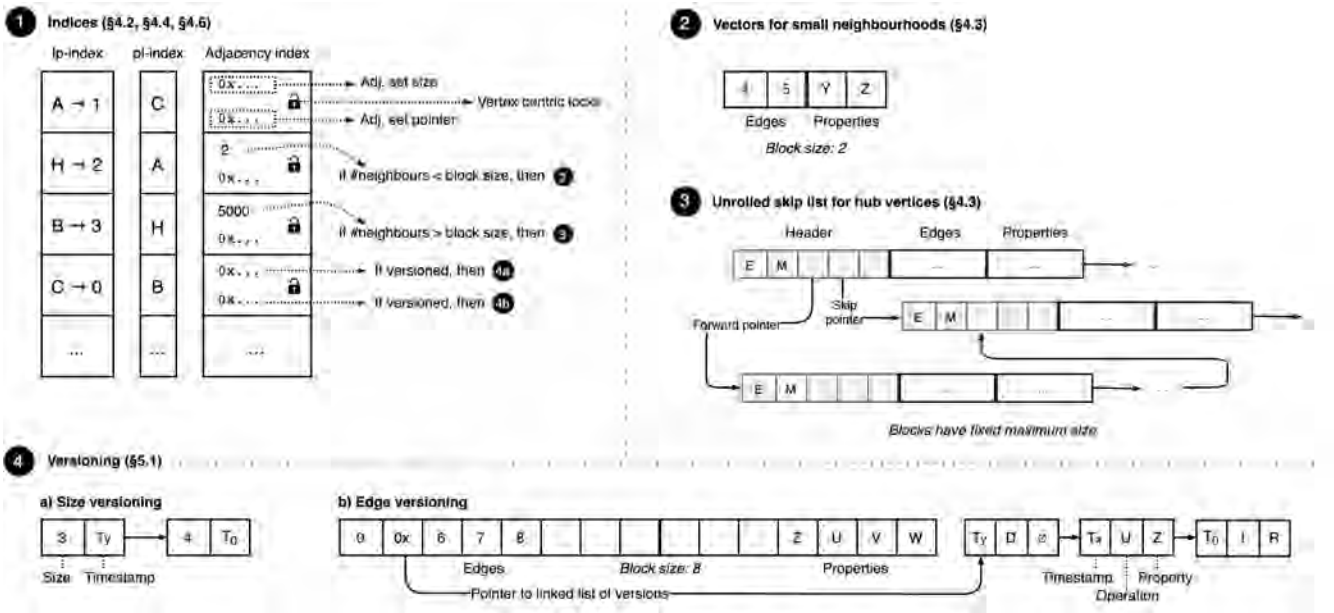


Figure 6: Sortledton's data structure for vertices H, A and B of the graph in Figure 2: ① Translation and *Adjacency Index*, ② Vectors for small neighborhoods, ③ unrolled skip list for hub vertices, ④ size and neighborhoods versioning.

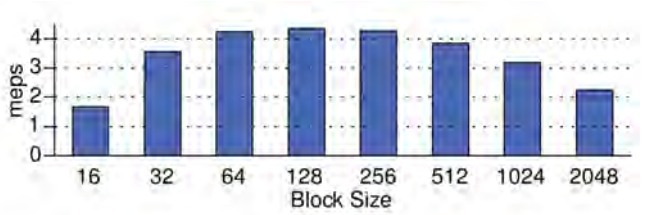


Figure 7: Sortledton's insertion throughput with varying block sizes.

durability, we propose using group commits, command *write-ahead logging*, and snapshotting as described thoroughly for relational workloads [38, 42, 43, 53], which can be implemented with low overhead [55]. However, we do not implement it because it is orthogonal to the other aspects of our design.

Version Storage: The version store aims to keep the memory overhead per edge as low as possible. An edge version records the type of the operation (insertion, update, or deletion), a commit timestamp, and the associated property. We store the version records as a linked list directly behind the edge in the block. The list is ordered from new to old versions. Versions of adjacency set sizes and vertices are stored similarly. Figure 6 ❹ shows an example: the user inserted $(3, 0)$ with property R, updated the property to Z and deleted it at timestamp T_x and T_y, respectively. The neighborhood size ❹ (a) of 3 has two versions: 4 at timestamp T_0 before $(3, 0)$ was deleted and 3 after the insertion at T_y. A linked list stores the edge versions for the deletion and the two property updates, the latest property value is also stored directly in the edge block.

We optimize for the case of two edge versions or less (Section 3.5). For a single version-edge, no version record is stored and the edge is assigned the implicit timestamp *first version* (T_0) for all operations. For two version-edges, we store the version record inline.

Concurrency Protocol: Our concurrency protocol handles the first and second requirements that were outlined in Section 3.5. It decouples reads from writes and optimizes for high throughput on writes with a priori known write set. It adapts MVCC with read-only optimization (ROMVCC) and two-phase locking. The read-write transaction protocol has five steps:

(1) claim all locks in a global order
(2) complete all reads with the newest versions and abort if the consistency guarantees are not fulfilled (Section 3.5)
(3) get a commit timestamp
(4) complete all writes using the commit timestamp
(5) release all locks

We leverage the a priori known read and write-sets to claim all locks in a global order at the beginning of the transaction [48]. Furthermore, as the protocol cannot abort after finishing the read step, no rollback logging is necessary.

For read-only transactions, we draw a commit timestamp. When reading a value, we first acquire a read lock, read the latest version before our commit-timestamp, and then release the read lock. Despite being pessimistic, the read locks are practically harmless as they are only held for the duration of one read operation. Furthermore, using locks for read-only transactions, allows more scalable implementations of read-write transactions. That is, it guarantees atomic commit without the overheads of other protocols (*e.g.*, an atomic commit and a validation phase, or drawing two timestamps for read-write transactions [34, 57]).

We give a proof sketch for the correctness of the protocol. Let $t1$ and $t2$ be transactions with commit timestamps $x1$ and $x2$. If both are read-write transactions, they are serializable by 2PL. Let

$t1$ be read-only and $t2$ be read-write. If $x1 < x2$, $t1$ cannot read any values written by $t2$ because $t2$ versions are written with timestamp $x2$. If $x2 < x1$, we can reason that $t2$ already holds all locks before $t1$ starts because the locking phase is completed before $t2$ gets its timestamp. Since $t2$ releases its locks only after completing, $t1$ needs to wait for the commit or abort of $t2$, when it tries to read any value written by $t2$. Hence, it reads all values of $t2$ after a commit or none after an abort.

Garbage Collection: We address garbage collection in two steps: (1) when can a version be collected, and (2) who collects them? We collect a version once no transaction can access it anymore. That is, we track the timestamps of all active transactions and collect all versions which are invisible to all active transactions as done in Hyper [10]. This is an important optimization to avoid long version chains in the presence of long-running transactions. Garbage is collected by the threads that execute write transactions. This leads to good data locality, requires no background threads and memory is freed where and when it is needed [16, 43].

6 IMPLEMENTATION

We implemented Sortledton by composing existing data structures. The *adjacency index* and the indices for translation use the `ConcurrentVector` and `ConcurrentHashMap` from Intel's *Threading Building Blocks* [47]. For our *adjacency sets*, we implemented the unrolled skip list from Platz et al. [45] with slight modifications as indicated in Section 4.2 and 4.3.

Our interface is similar to prior work [19, 34] with two exceptions. First, we allow the user to access both the sparse and the dense vertex identifiers. We run graph computations by translating inputs into the dense domain, then running the analytics, and finally translating the output back to the original sparse domain. We show this process in Listing 2. The translation is fast because the input to most computations is small while the output translation is cheap and easy to parallelize. For example, a BFS receives its start ID as input and outputs one distance value per vertex. So, we need $V+1$ parallel translations, which take only a fraction of the runtime.

Data: tx: *ReadOnlyTransaction*, start_vertex: *logical_id*
Result: result: *array of size V containing the hop distance from* `start_vertex`

```
1 start_vertex_dense ← tx.dense_id(start_vertex)
2 distances ← bfs(start_vertex_dense)
3 for i ← 0 to V do
4 |   result[i] ← pair(tx.sparse_id(i), distances[i])
```

Algorithm 2: Running a BFS on a dense domain using sparse identifiers externally.

Second, we optimize the interface that scans neighborhoods. Prior works offer three different methods to access their neighborhoods: (1) via an iterator interface [19, 57], (2) by providing a lambda function executed for each edge [34, 52], or (3) by direct memory access (*e.g.*, for the CSR). However, using an iterator or a lambda function executes two or one function(s) per edge, respectively. To avoid this, we allow direct memory access to blocks without any versions. In Figure 8, we present a comparison of graph computations with and without this optimization. It can lead to speedups of up to 2.3x.

Figure 8: Slowdown when using an iterator interface instead of direct memory access.

7 EVALUATION

We run our experiments on a dual-socket machine with Intel Xeon E5-2680v4 processors, which has 70 MiB of L3 cache, 14 hardware threads, and 256 GB of memory. We compiled all systems with GCC v10.2 and the O3 parameter. Further, we disabled Linux's NUMA aware page migration feature. Numbers reported are the median of 5 runs. For Graphalytics kernels, runtimes include translation costs for inputs and outputs for all systems but Teseo on sparse identifiers. We use a state-of-the-art kernel implementation (see Section 2.3) and we disable disk-logging for all systems. For Sortledton, we set the *block size* to 512 trading insertion for analytical performance (cf . Figure 4b and Figure 7). We add software prefetching to BFS as described in Section 3.2.

7.1 Qualitative Comparison to Related Work

We compare our work to a diverse range of state-of-the-art dynamic graph data structures that support single edge updates: Stinger [19], GraphOne [30], LLama [36], Livegraph [57], and Teseo [34]. We relate the data structures used by all systems with the memory access patterns (Section 2.2) and the high-level designs (Section 4.1).

Stinger, GraphOne, and Livegraph are *adjacency list-like*. Hence, they have good support for *random vertex access* and are not optimized for *sequential vertex access*. Livegraph uses one vector per neighborhood for optimal *sequential neighborhood access*. Stinger and GraphOne use one (14 edges) and two fixed block sizes (8 or 512 edges) for their neighborhoods, respectively. All of them use neighborhood data structures that appenḥe inserts. Hence, they achieve good isolation of writing and reading threads for concurrency control of read-only queries. However, this does not allow for efficient graph pattern matching.

LLama and Teseo have a *CSR-like* design and optimize for *sequential vertex access*. LLama's read-store holds multiple snapshots. A snapshot is a sorted array of new edges since the last snapshot. Writes are buffered in a key-value store. We create a new snapshot every 10 seconds as suggested by the authors [36]. The snapshotting fragments neighborhoods. Hence, LLama does not optimize for *sequential neighborhood access*. However, this is hidden due to temporal locality when the computation follows the *sequential vertex access pattern*. The combination of *random vertex access* and *sequential neighborhood access* is most challenging for LLama.

Teseo stores its vertices and edges in a B+ tree with 2MB-sized leaves containing packed memory arrays [33]. Packed memory

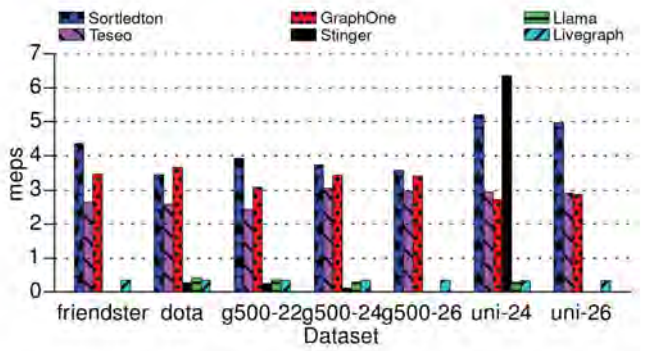


Figure 9: Edge insertion throughput with random edge ordering. GraphOne upholds lower consistency guarantees.

arrays store blocks of elements interleaved with gaps to allow insertions. Blocks contain multiple neighborhoods and up to 512 edges. Thus, Teseo has good support for *sequential vertex access* and *sequential neighborhood access*. For *random vertex access*, Teseo uses a hash map that is lazily updated. Teseo is designed to store sparse vertex identifiers and needs to compute a dense mapping to interface with existing graph algorithms. They need to translate each edge read during analytics into the dense domain. Since this is expensive, the authors provide a specialized version that can load only graphs with dense identifiers. For graph computations, we measure both versions.

7.2 Insertions Performance

The experiment evaluates the throughput of single edge insertions in all systems. We add all edges of the input graph in random order, one-by-one as undirected edges with no sleep time on the user side. The addition of an edge checks if both vertices exist and inserts them if not. The next step asserts that the edge does not exist before inserting it. Livegraph, Teseo, and Sortledton perform all operations for each edge insertion encapsulated in a transaction, thereby ensuring atomicity and isolation. The other systems give no guarantees and GraphOne cannot check if an edge already exists.

Figure 9 shows the throughput in million edges per second (meps). Missing bars indicate that a system could not load the graph due to memory restrictions. For power-law graphs (first 6), Teseo and Sortledton are superior to the others because their neighborhood sets allow for efficient checks if an edge exists. GraphOne has similar performance, but as noted earlier does not perform the check if an edge exists. If we introduce this check, its throughput would drop to 5 edges per second [34]. Sorltedton's processes up to 1.6 million edges per second more than Teseo without using background threads while using less memory (see Section 7.3) and versioning adjacency set sizes.

For uniform graphs (first 2 from the right), Stinger demonstrates the best performance – although, it cannot load *uni-26* on our system due to its high memory consumption. This reveals that for uniform graphs, it is cheaper to linearly search for the existence of an edge than paying the price of keeping them sorted.

Resilience to real-world update patterns: Until now, we load all edges in random order to be comparable with past work [30, 34, 36]. However, the *yahoo-songs* and *edit-wiki* graphs show a strong temporal locality between updates to the same vertex. In this experiment, we load these graphs in the order of their edge creation timestamps with no sleep time as opposed to the actual update frequency of a few edges per minute.

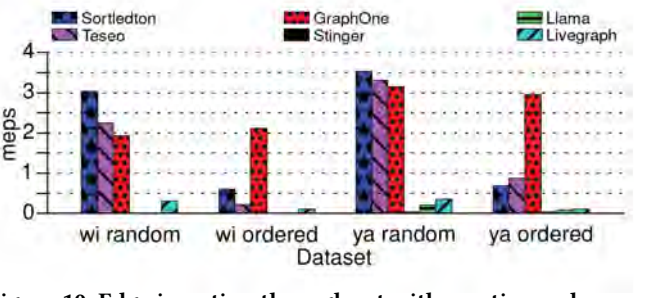


Figure 10: Edge insertion throughput with creation order.

Figure 10 shows the throughput for both creation and random order of applying the updates. Sortledton, Livegraph, and LLama, have lower throughput when the graph is loaded in creation order because they use one lock per vertex leading to higher contention in bursty workloads. Although Teseo can split large neighborhoods over multiple blocks, it suffers from higher contention when loading in sorted order. GraphOne and Stinger are not affected because they batch updates or use lock-free synchronization, respectively. Stinger and LLama cannot load *edit-wiki* in either order. It is a bipartite graph with 5 times more vertices in one partition, a high edge to vertex ratio of 5:1 and a high maximum degree of 5M edges.

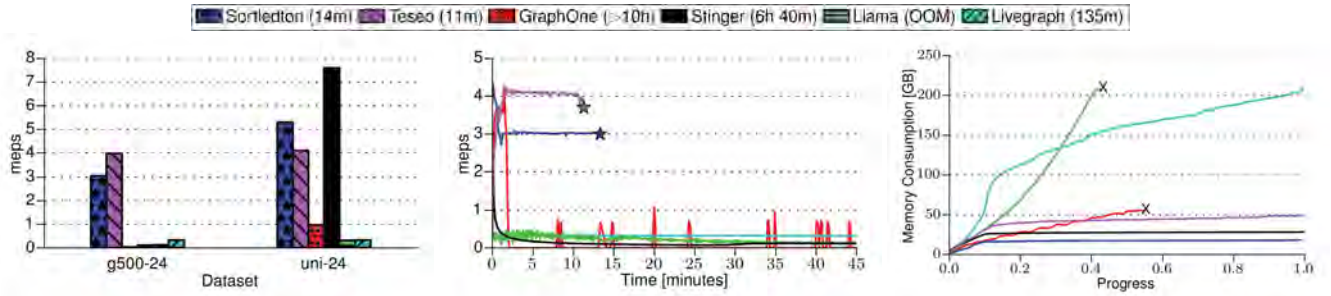
We conclude that vertex-centric locking is a weak point for bursty workloads and should be replaced by lock-free synchronization. Furthermore, benchmarks should specify the loading order of edges, because it can significantly influence the system performance and they should cover the complete space of graph types, *i.e.*, Graphalytics and GAPBS do not include a bipartite graph [6, 11].

7.3 Updates

We next evaluate how the data structures behave when running a balanced mix of insertions and deletions with the same setup as in Teseo [34] on an already large graph. The experiment has two phases. The first 10% of all operations load *Graph500-24*. After, we run a balanced mix 9x the operations as insertions and deletions while keeping the graph size stable. We discuss throughput over time, average throughput, and memory usage.

Average throughput (Figure 11a) allows us to compare how the different data structures react to deletions by comparing their throughput to the insertion-only experiment (Section 7.2). Livegraph and Stinger show no significant difference because they treat insertions and deletions in the same way. Sortledton's throughput is slightly lower due to the creation and maintenance of version chains. Teseo profits from deletions as they free space and lead to fewer rebalances. LLama's throughput halves compared to insertions-only and GraphOne's is 64 times lower.

Throughput over time (Figure 11b) shows how the systems react to a growing workload and the effects of snapshotting in LLama and GraphOne. The legend shows the completion time per system. Teseo and Sortledton finish within 14 minutes while all other systems take more than 2 hours.



(a) Average throughput. **(b) Average througput per second for *g500-24*.** **(c) Memory consumption for *g500-24*.**

Figure 11: Throughput and memory consumptions on a mixture of insertions and deletions. Total execution time in the legend.

For Sorteldton, Teseo, and Livegraph, we observe smooth lines as they perform updates individually and in place. They do not depend on the current size of the dataset. Stinger loses performance while the data structure grows, but has a stable throughput afterwards. LLama's throughput decreases over time because of neighborhood fragmentation and the memory pressure by storing multiple snapshots. Furthermore, their throughput oscillates due to the need to digest edges from write to read-store. GraphOne shows throughput between 0 and 1 million edges per second. The system becomes unresponsive when applying edges to its read store because it struggles to locate edges to delete.

Memory consumption in Figure 11c allows us to categorize the systems into two classes. Livegraph, LLama, and GraphOne show increasing memory consumption due to partial or missing garbage collection implementations. Teseo's, Stinger's, and Sortledton's memory usage grow until the graph reaches its final size at 10% of all operations. Then they show stable memory consumption. They use 48 GB (Teseo), 27 GB (Stinger), and 18 GB (Sortledton), respectively. For reference, storing *graph500-24* statically in a CSR requires 8.3 GB. So, Sortledton's overhead is 2.1x because our blocks are 75% full on average, and we store the vertex ID translations in $2 \times |V|$. The first overhead is inherent in many dynamic data structures (*e.g.*, B-trees, vectors, or hash sets). The second is needed by systems that offer a sparse domain and a dense domain.

This experiment leads us to three conclusions. First, list-based designs struggle with deletions. Second, batching updates in a write-optimized store leads to high average throughput but comes at the cost of unstable throughput and can lead to unresponsiveness. Third, it is possible to store a dynamic graph in twice the memory needed for a static graph.

7.4 Multicore Scalability

Figure 12 shows multicore scalability from 1 to 56 threads for all systems on *graph500-24*. Teseo and Sortledton execute more than 3.1 and 4.8 million checked edge insertions per second, respectively. Most other systems achieve less than 500 thousand. GraphOne achieves 3.4 million unchecked insertions per second.

Both Teseo and Sortledton scale up to 56 threads. Sortledton scales better because its concurrency control protocol is more lightweight. One could achieve even better scalability with Sortledton by using contention-free hash sets for translation between the vertex domains or adding NUMA-awareness.

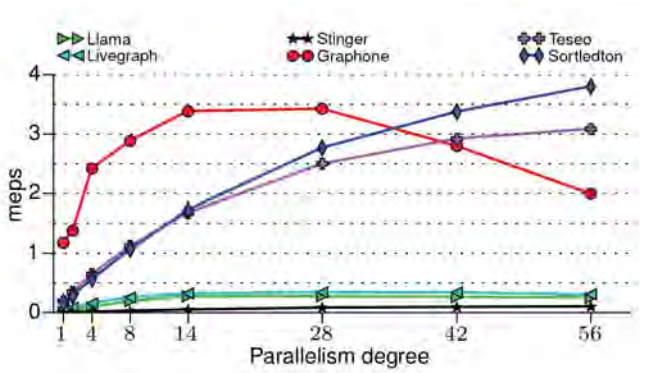


Figure 12: Edge insertion throughput and multicore scalability on *graph500-24*.

GraphOne does not scale beyond 14 threads because vertex ID translation is sequential, and due to contention on its write buffer [34]. However, it is twice as efficient with 14 threads as Teseo and Sortledton. This is because (1) the different insertion semantics, and (2) GraphOne batches updates in a circular buffer, then partitions them per vertex and applies the partitions in parallel to the main data structure. While this design enables high throughput, it leads to higher update latencies and requires building a snapshot before analytics. Therefore, it is not suited for Alibaba's fraud detection workload with strict latency requirements and computations per edge insertion [46, 56].

7.5 Analytics

We analyze the influence of the different data structures on the runtime of analytics, traversals, and GPM queries. We run the Graphalytics benchmark kernels as defined in Section 2.3 after loading the graph edge-by-edge. We normalize the runtimes against using a CSR, which is arguably the best general-purpose baseline for static graphs. When beneficial, we use a NUMA optimized CSR as baseline.

Figure 13 shows the slowdown of each system. We select *dota-league*, *Graph500-24*, and *com-friendster* as a representative set of graphs. Missing bars either indicate that the data structure could not load the graph due to the memory constraints, or did not complete the kernel computation within an hour.

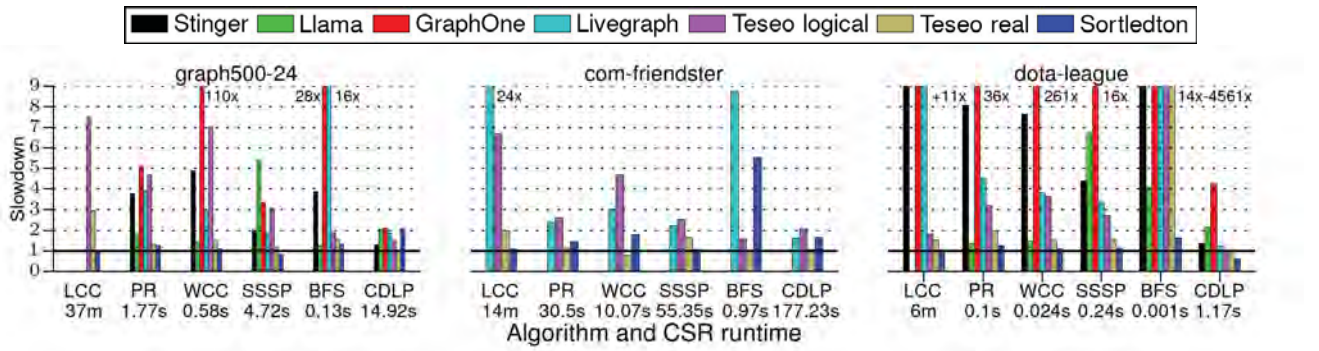


Figure 13: Graph kernel runtimes normalized to CSR.

LCC, a GPM algorithm, shows no slowdown for Sortledton, a 3x for Teseo and 11x to 106x or no completion for other systems. This is due to Teseo's and Sortledton's set-based neighborhoods.

WCC and PR exhibit the *sequential vertex access* and *sequential neighborhood patterns*. Sortledton, Teseo on dense vertices, and LLama have runtimes close to the CSR, confirming that support for either pattern is effective. Teseo on sparse vertices takes up to 14x longer because it translates each edge into the dense domain using a hashmap lookup (Section 2.2). Stinger's small, fixed-size neighborhood blocks lead to pointer chasing (cf. Figure 4b). Livegraph needs to scan 3x as much data and evaluates a predicate for each scanned edge. GraphOne implements access to its neighborhood by copying them into a user-provided vector.

SSSP combines *random vertex access* with *sequential neighborhoods accesses*. The runtime for all systems, apart from LLama, is similar to WCC and PR. LLama does not optimize for *sequential neighborhood access* in combination with *random vertex access*.

BFS shows the highest variance among all systems. We use the direction-optimized variant by Scott Beamer [5]. It exhibits the *sequential vertex* and the *sequential neighborhood access patterns*. It differs from all other kernels, as it stops scanning early after finding an edge that satisfies a predicate; often scanning less than 8 edges. Hence, Livegraph and GraphOne have decreased performance due to their overheads for accessing a few edges. On *graph500-24*, other systems show similar performance as for PR and WCC. However, slowdowns on *com-friendster* or *dota-league* show that BFS is hard to optimize for. CDLP is dominated by building histograms of IDs. Hence, it is not indicative for the graph data structure performance.

7.6 Concurrent Read-Write Workload

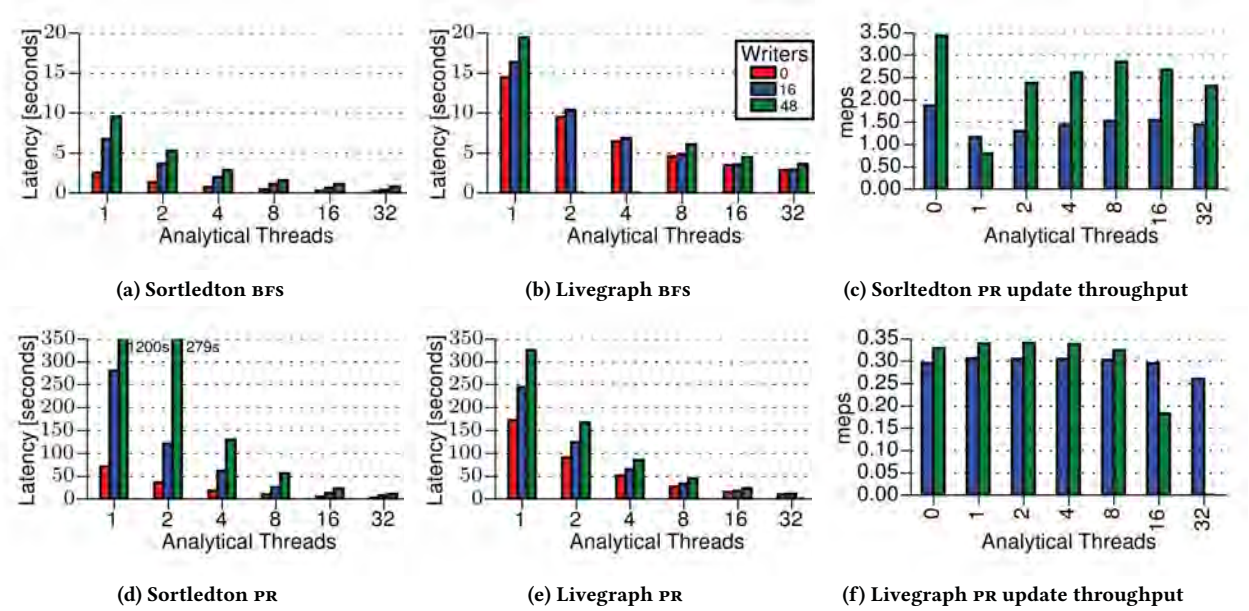
We evaluate the influence of concurrently executing updates from Section 7.3 and BFS or PR from Section 7.5. Figure 14 shows the latency of analytics and the throughput of updates when combining 1 to 32 analytical threads with 16 and 48 writers compared to running them in isolation. The experiment can only be executed by transactional systems. Teseo cannot execute the workload due to a bug.

When BFS is run concurrently with updates, the throughput is at most 12% (Sortledton) and 22% (Livegraph) lower than updates in isolation. When updates run alongside PR, the throughput drops significantly (Figure 14c). There are two reasons for this: 1) PR scans complete neighborhoods while BFS scans only parts of most neighborhoods, 2) PR runs longer transactions. Consequently, PR holds locks on neighborhoods longer and leads to a higher number of versions in the system. In particular, a long PR query with 1 analytical thread strains the system with many versions [10]. The latency of both BFS and PR is higher than in isolation (Figures 14a and 14d), due to multi-versioned edges which disable our direct access optimization (Figure 8) and instead follow version chains.

For Livegraph, the concurrent workloads affect each other less. There are two reasons: 1) the use of a log-structured data structure that allows writers to append the inserts without affecting the readers. 2) Livegraph always pays the overhead of having all edges versioned which lowers efficiency. Despite the interference, Sortledton is ahead for updates/analytics in all/most cases.

8 RELATED WORK

We discuss graph data structures with support for single edge inserts [19, 30, 34, 36, 57]. Figure 1 relates them to the challenges outlined in Section 2.1. Only, Teseo and Sortledton solve the first challenge of supporting graph pattern matching by computing intersections in linear time. The second challenge of allowing for concurrent updates and computations is only addressed by Teseo, Livegraph, and Sortledton. Livegraph's concurrency control is based on an OLTP-optimized protocol [32], that causes overheads on analytical workloads and memory consumption. Teseo's protocol is an HTAP-optimized protocol [43] for general read-write transactions. This comes at the cost of having a higher overhead on small transactions with a known write set. In particular, they have a higher abort rate, do rollback logging, need to draw two timestamps per transaction, and have a sequential validation phase. As a result, only Teseo and Sorltedton address both challenges, but they follow fundamentally different designs. Sortledton has an *adjacency list-like* design, while Teseo follows the *csr-like* design. We compared these designs in Section 4.1. Only Sortledton and Livegraph could execute analytics and updates concurrently. LLama, GraphOne and Livegraph optionally support disk-based storage. We discuss three other differentiators between Sortledton and related work. First, GraphOne, LLama, and Teseo use read and write-optimized segments to handle inserts. This leads to lower read performance or reduced freshness. In the case of LLama and GraphOne, this will result in unstable throughput over time (cf. Figure 11b). Second, all competitors rely on background threads to perform data structure maintenance (*e.g.* Teseo uses one thread per core for rebalancing and garbage collection [34]). In particular, in combination with



(a) Sortledton BFS (b) Livegraph BFS (c) Sorltedton PR update throughput

(d) Sortledton PR (e) Livegraph PR (f) Livegraph PR update throughput

Figure 14: Mixed workload for Graph500-24.

compute-intensive GPM this leads to overprovisioning. Finally, most systems run pre-computations before analytics after applying updates [30, 34, 36]. LLama and GraphOne ingest all buffered writes into read-optimized storage. Similarly, GraphOne and Teseo create a snapshot in $O(V)$ steps before starting analytics. GraphOne stores the sizes of the neighborhoods to guarantee isolation from new updates. Teseo creates a translation from sparse to dense vertices.

Batched update graph data structures trade-off update latency for higher throughput. Aspen and Terrace support fast scans and intersections [15, 44]. Aspen is *adjacency list-like* and can run coarse-grained transactions per update batch by a single-writer copy-on-write scheme. It uses purely functional trees storing blocks of edges and a functional tree for the adjacency index. Terrace mixes a *adjacency list-like* and *CSR-like* design. It uses three different data structures depending on the size of the neighborhoods: they inline small neighborhoods in the index, use packed memory arrays for medium-sized neighborhoods, and B-Trees for hub vertices.

Graph Databases. Mature graph databases exist, *e.g.*, Neo4J and Virtuoso. They use a linked list of edges per vertex and a columnar relational layout for storage. Neo4J's concurrency control uses the isolation level read-committed and Virtuoso uses single version locking. Hence, both systems could profit by changing to Sortledton as underlying storage because of its cache-friendly layout and low neighborhood lookup latencies as well as the higher isolation level and/or better decoupling of readers and writers, respectively.

Further graph workloads. So far, we have discussed analytics, traversals, and graph pattern matching workloads because they drive our design. However, Besta et al. list three further workloads: local, neighborhood, and the LDBC interactive and business intelligence benchmarks. These can be efficiently supported by our low-latency index and ability to find existing edges. For the LDBC workload, like other dynamic data structures, we do not support labels. For static use cases, the issue is addressed by Mhedhbi et al. [40].

9 CONCLUSIONS

Sortledton is a sorted, simple, transactional graph data structure that executes up to 5 million edge updates per second, supports analytical, GPM, and traversal workloads with runtimes within 1.2x on average of CSR while needing only ~2x the space of CSR. Furthermore, it runs analytics and a high number of updates concurrently. We achieve this by reusing existing data structures.

We construct Sortledton based on two key principles. First, a universal graph data structure needs to store neighborhoods in sets to support GPM, consistency, edge updates, and deletions. Second, we identify four memory access patterns in graph workloads: *sequential vertex*, *sequential neighborhood*, *algorithmic-specific property*, and *random vertex access patterns*. With a series of microbenchmarks, we show that it is more important to optimize for *sequential neighborhood access* and *algorithmic-specific property access* because they occur once per edge, rather than the other two access patterns that occur once per vertex. Therefore, *csr-like* designs lose their main advantage over *adjacency list-based* designs that are significantly simpler to build.

ACKNOWLEDGMENTS

We thank Dean De Leo for taking a big step towards making graph data structure research comparable with his test driver. We thank Maximilian Bandle, Dominik Durner, and our reviewers for their valuable feedback.

References

[1] Christopher R. Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. 2016. EmptyHeaded: A Relational Engine for Graph Processing. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. ACM, 431–446. https://doi.org/10.1145/2882903.2915213

[2] Renzo Angles, János Benjamin Antal, Alex Averbuch, Peter A. Boncz, Orri Erling, Andrey Gubichev, Vlad Haprian, Moritz Kaufmann, Josep Lluís Larriba-Pey, Norbert Martínez-Bazan, József Marton, Marcus Paradies, Minh-Duc Pham, Arnau Prat-Pérez, Mirko Spasic, Benjamin A. Steer, Gábor Szárnyas, and Jack Waudby. 2020. The LDBC Social Network Benchmark. *CoRR* abs/2001.02299 (2020). arXiv:2001.02299 http://arxiv.org/abs/2001.02299

[3] Renzo Angles, Marcelo Arenas, Pablo Barceló, Peter A. Boncz, George H. L. Fletcher, Claudio Gutiérrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan F. Sequeda, Oskar van Rest, and Hannes Voigt. 2018. G-CORE: A Core for Future Graph Query Languages. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. ACM, 1421–1432. https://doi.org/10.1145/3183713.3190654

[4] Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. 2013. LinkBench: a database benchmark based on the Facebook social graph. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*. ACM, 1185–1196. https://doi.org/10.1145/2463676.2465296

[5] Scott Beamer, Krste Asanovic, and David A. Patterson. 2013. Direction-optimizing breadth-first search. *Sci. Program.* 21, 3-4 (2013), 137–148. https://doi.org/10.3233/SPR-130370

[6] Scott Beamer, Krste Asanovic, and David A. Patterson. 2015. The GAP Benchmark Suite. *CoRR* abs/1508.03619 (2015). arXiv:1508.03619 http://arxiv.org/abs/1508.03619

[7] Scott Beamer, Krste Asanovic, and David A. Patterson. 2015. Locality Exists in Graph Processing: Workload Characterization on an Ivy Bridge Server. In *2015 IEEE International Symposium on Workload Characterization, IISWC 2015, Atlanta, GA, USA, October 4-6, 2015*. IEEE Computer Society, 56–65. https://doi.org/10.1109/IISWC.2015.12

[8] Maciej Besta, Emanuel Peter, Robert Gerstenberger, Marc Fischer, Michal Podstawski, Claude Barthels, Gustavo Alonso, and Torsten Hoefler. 2019. Demystifying Graph Databases: Analysis and Taxonomy of Data Organization, System Designs, and Graph Queries. *CoRR* abs/1910.09017 (2019). arXiv:1910.09017 http://arxiv.org/abs/1910.09017

[9] Jan Böttcher, Viktor Leis, Jana Giceva, Thomas Neumann, and Alfons Kemper. 2020. Scalable and robust latches for database systems. In *16th International Workshop on Data Management on New Hardware, DaMoN 2020, Portland, Oregon, USA, June 15, 2020*. ACM, 2:1–2:8. https://doi.org/10.1145/3399666.3399908

[10] Jan Böttcher, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2019. Scalable Garbage Collection for In-Memory MVCC Systems. *Proc. VLDB Endow.* 13, 2 (2019), 128–141. https://doi.org/10.14778/3364324.3364328

[11] Mihai Capota, Tim Hegeman, Alexandru Iosup, Arnau Prat-Pérez, Orri Erling, and Peter A. Boncz. 2015. Graphalytics: A Big Data Benchmark for Graph-Processing Platforms. In *Proceedings of the Third International Workshop on Graph Data Management Experiences and Systems, GRADES 2015, Melbourne, VIC, Australia, May 31 - June 4, 2015*. ACM, 7:1–7:6. https://doi.org/10.1145/2764947.2764954

[12] Shumo Chu, Magdalena Balazinska, and Dan Suciu. 2015. From Theory to Practice: Efficient Join Query Evaluation in a Parallel Database System. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. ACM, 63–78. https://doi.org/10.1145/2723372.2750545

[13] Douglas Comer. 1979. The Ubiquitous B-Tree. *ACM Comput. Surv.* 11, 2 (1979), 121–137. https://doi.org/10.1145/356770.356776

[14] Damian Dechev, Peter Pirkelbauer, and Bjarne Stroustrup. 2006. Lock-Free Dynamically Resizable Arrays. In *Principles of Distributed Systems, 10th International Conference, OPODIS 2006, Bordeaux, France, December 12-15, 2006, Proceedings (Lecture Notes in Computer Science)*, Vol. 4305. Springer, 142–156. https://doi.org/10.1007/11945529_11

[15] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2019. Low-latency graph streaming using compressed purely-functional trees. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 918–934. https://doi.org/10.1145/3314221.3314598

[16] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*. ACM, 1243–1254. https://doi.org/10.1145/2463676.2463710

[17] Bolin Ding, Kai Zeng, and Wenyuan Yu. 2020. Alibaba Sponsor Talk at VLDB.

[18] Vinícius Vitor dos Santos Dias, Carlos H. C. Teixeira, Dorgival O. Guedes, Wagner Meira Jr., and Srinivasan Parthasarathy. 2019. Fractal: A General-Purpose Graph Pattern Mining System. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*. ACM, 1357–1374. https://doi.org/10.1145/3299869.3319875

[19] David Ediger, Robert McColl, E. Jason Riedy, and David A. Bader. 2012. STINGER: High performance data structure for streaming graphs. In *IEEE Conference on High Performance Extreme Computing, HPEC 2012, Waltham, MA, USA, September 10-12, 2012*. IEEE, 1–5. https://doi.org/10.1109/HPEC.2012.6408680

[20] Assaf Eisenman, Ludmila Cherkasova, Guilherme Magalhaes, Qiong Cai, Paolo Faraboschi, and Sachin Katti. 2016. Parallel Graph Processing: Prejudice and State of the Art. In *Proceedings of the 7th ACM/SPEC International Conference on Performance Engineering, ICPE 2016, Delft, The Netherlands, March 12-16, 2016*. ACM, 85–90. https://doi.org/10.1145/2851553.2851572

[21] Orri Erling, Alex Averbuch, Josep Lluís Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat-Pérez, Minh-Duc Pham, and Peter A. Boncz. 2015. The LDBC Social Network Benchmark: Interactive Workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. ACM, 619–630. https://doi.org/10.1145/2723372.2742786

[22] Per Fuchs, Peter A. Boncz, and Bogdan Ghit. 2020. EdgeFrame: Worst-Case Optimal Joins for Graph-Pattern Matching in Spark. In *GRADES-NDA'20: Proceedings of the 3rd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA), Portland, OR, USA, June 14, 2020*. ACM, 4:1–4:11. https://doi.org/10.1145/3398682.3399162

[23] Alastair Green. 2019. *THE GQL MANIFESTO*. https://gql.today/

[24] Pranjal Gupta, Amine Mhedhbi, and Semih Salihoglu. 2021. Integrating Column-Oriented Storage and Query Processing Techniques Into Graph Database Management Systems. *CoRR* abs/2103.02284 (2021). arXiv:2103.02284 https://arxiv.org/abs/2103.02284

[25] Pankaj Gupta, Venu Satuluri, Ajeet Grewal, Siva Gurumurthy, Volodymyr Zhabiuk, Quannan Li, and Jimmy J. Lin. 2014. Real-Time Twitter Recommendation: Online Motif Detection in Large Dynamic Graphs. *Proc. VLDB Endow.* 7, 13 (2014), 1379–1380. https://doi.org/10.14778/2733004.2733010

[26] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. 2007. A Simple Optimistic Skiplist Algorithm. In *Structural Information and Communication Complexity, 14th International Colloquium, SIROCCO 2007, Castiglioncello, Italy, June 5-8, 2007, Proceedings (Lecture Notes in Computer Science)*, Vol. 4474. Springer, 124–138. https://doi.org/10.1007/978-3-540-72951-8_11

[27] Stratos Idreos, Kostas Zoumpatianos, Manos Athanassoulis, Niv Dayan, Brian Hentschel, Michael S. Kester, Demi Guo, Lukas M. Maas, Wilson Qin, Abdul Wasay, and Yiyou Sun. 2018. The Periodic Table of Data Structures. *IEEE Data Eng. Bull.* 41, 3 (2018), 64–75. http://sites.computer.org/debull/A18sept/p64.pdf

[28] SQL ISO. 2020. *ISO/IEC CD 9075-16*. https://www.iso.org/standard/79473.html

[29] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. 2009. Optimistic parallelism requires abstractions. *Commun. ACM* 52, 9 (2009), 89–97. https://doi.org/10.1145/1562164.1562188

[30] Pradeep Kumar and H. Howie Huang. 2019. GraphOne: A Data Store for Real-time Analytics on Evolving Graphs. In *17th USENIX Conference on File and Storage Technologies, FAST 2019, Boston, MA, February 25-28, 2019*. USENIX Association, 249–263. https://www.usenix.org/conference/fast19/presentation/kumar

[31] Jérôme Kunegis. 2013. KONECT: the Koblenz network collection. In *22nd International World Wide Web Conference, WWW '13, Rio de Janeiro, Brazil, May 13-17, 2013, Companion Volume*. International World Wide Web Conferences Steering Committee / ACM, 1343–1350. https://doi.org/10.1145/2487788.2488173

[32] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. 2011. High-Performance Concurrency Control Mechanisms for Main-Memory Databases. *Proc. VLDB Endow.* 5, 4 (2011), 298–309. https://doi.org/10.14778/2095686.2095689

[33] Dean De Leo and Peter A. Boncz. 2019. Fast Concurrent Reads and Updates with PMAs. In *Proceedings of the 2nd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA), Amsterdam, The Netherlands, 30 June 2019*. ACM, 8:1–8:8. https://doi.org/10.1145/3327964.3328497

[34] Dean De Leo and Peter A. Boncz. 2021. Teseo and the Analysis of Structural Dynamic Graphs. *Proc. VLDB Endow.* 14, 6 (2021), 1053–1066. http://www.vldb.org/pvldb/vol14/p1053-leo.pdf

[35] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[36] Peter Macko, Virendra J. Marathe, Daniel W. Margo, and Margo I. Seltzer. 2015. LLAMA: Efficient graph analytics using Large Multiversioned Arrays. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*. IEEE Computer Society, 363–374. https://doi.org/10.1109/ICDE.2015.7113298

[37] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*. ACM, 135–146. https://doi.org/10.1145/1807167.1807184

[38] Nirmesh Malviya, Ariel Weisberg, Samuel Madden, and Michael Stonebraker. 2014. Rethinking main memory OLTP recovery. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, Isabel F. Cruz, Elena Ferrari, Yufei Tao, Elisa Bertino, and Goce Trajcevski (Eds.). IEEE Computer Society, 604–615. https://doi.org/10.1109/ICDE.2014.6816685

[39] Mugilan Mariappan and Keval Vora. 2019. GraphBolt: Dependency-Driven Synchronous Processing of Streaming Graphs. In *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*. ACM, 25:1–25:16. https://doi.org/10.1145/3302424.3303974

[40] Amine Mhedhbi, Pranjal Gupta, Shahid Khaliq, and Semih Salihoglu. 2020. A+ Indexes: Lightweight and Highly Flexible Adjacency Lists for Graph Database Management Systems. *CoRR* abs/2004.00130 (2020). arXiv:2004.00130 https://arxiv.org/abs/2004.00130

[41] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins. *Proc. VLDB Endow.* 12, 11 (2019), 1692–1704. https://doi.org/10.14778/3342263.3342643

[42] C. Mohan, Don Haderle, Bruce G. Lindsay, Hamid Pirahesh, and Peter M. Schwarz. 1992. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Trans. Database Syst.* 17, 1 (1992), 94–162. https://doi.org/10.1145/128765.128770

[43] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. ACM, 677–689. https://doi.org/10.1145/2723372.2749436

[44] Prashant Pandey, Brian Wheatman, Helen Xu, and Aydin Buluç. 2021. Terrace: A Hierarchical Graph Container for Skewed Dynamic Graphs. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 1372–1385. https://doi.org/10.1145/3448016.3457313

[45] Kenneth Platz, Neeraj Mittal, and S. Venkatesan. 2019. Concurrent Unrolled Skiplist. In *39th IEEE International Conference on Distributed Computing Systems, ICDCS 2019, Dallas, TX, USA, July 7-10, 2019*. IEEE, 1579–1589. https://doi.org/10.1109/ICDCS.2019.00157

[46] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. 2018. Real-time Constrained Cycle Detection in Large Dynamic Graphs. *Proc. VLDB Endow.* 11, 12 (2018), 1876–1888. https://doi.org/10.14778/3229863.3229874

[47] James Reinders. 2007. *Intel threading building blocks - outfitting C++ for multi-core processor parallelism*. O'Reilly. http://www.oreilly.com/catalog/9780596514808/index.html

[48] Kun Ren, Jose M. Faleiro, and Daniel J. Abadi. 2016. Design Principles for Scaling Multi-core OLTP Under High Contention. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. ACM, 1583–1598. https://doi.org/10.1145/2882903.2882958

[49] Yousef Saad. 2003. *Iterative methods for sparse linear systems*. SIAM. https://doi.org/10.1137/1.9780898718003

[50] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. 2017. The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing. *Proc. VLDB Endow.* 11, 4 (2017), 420–431. https://doi.org/10.1145/3186728.3164139

[51] Aneesh Sharma, Jerry Jiang, Praveen Bommannavar, Brian Larson, and Jimmy J. Lin. 2016. GraphJet: Real-Time Content Recommendations at Twitter. *Proc. VLDB Endow.* 9, 13 (2016), 1281–1292. https://doi.org/10.14778/3007263.3007267

[52] Julian Shun and Guy E. Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, Shenzhen, China, February 23-27, 2013*. ACM, 135–146. https://doi.org/10.1145/2442516.2442530

[53] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*. ACM, 18–32. https://doi.org/10.1145/2517349.2522713

[54] Keval Vora, Rajiv Gupta, and Guoqing Xu. 2017. KickStarter: Fast and Accurate Computations on Streaming Graphs via Trimmed Approximations. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*. ACM, 237–251. https://doi.org/10.1145/3037697.3037748

[55] Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov. 2014. Fast Databases with Fast Durability and Recovery Through Multicore Parallelism. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, Jason Flinn and Hank Levy (Eds.). USENIX Association, 465–477. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/zheng_wenting

[56] Xiaowei Zhu, Zhisong Fu, Zhenxuan Pan, Jin Jiang, Chuntao Hong, Yongchao Liu, Yang Fang, Wenguang Chen, and Changhua He. 2021. Taking the Pulse of Financial Activities with Online Graph Processing. *ACM SIGOPS Oper. Syst. Rev.* 55, 1 (2021), 84–87. https://doi.org/10.1145/3469379.3469389

[57] Xiaowei Zhu, Marco Serafini, Xiaosong Ma, Ashraf Aboulnaga, Wenguang Chen, and Guanyu Feng. 2020. LiveGraph: A Transactional Graph Storage System with Purely Sequential Adjacency List Scans. *Proc. VLDB Endow.* 13, 7 (2020), 1020–1034. https://doi.org/10.14778/3384345.3384351

NBTree: a Lock-free PM-friendly Persistent B$^+$-Tree for eADR-enabled PM Systems

Bowen Zhang
Shanghai Jiao Tong University
bowenzhang@sjtu.edu.cn

Shengan Zheng
MoE Key Lab of Artificial Intelligence, AI Institute,
Shanghai Jiao Tong University
venero1209@sjtu.edu.cn

Zhenlin Qi
Shanghai Jiao Tong University
qizhenlin@sjtu.edu.cn

Linpeng Huang
Shanghai Jiao Tong University
lphuang@sjtu.edu.cn

ABSTRACT

Persistent memory (PM) promises near-DRAM performance as well as data persistency. Recently, a new feature called eADR is available on the 2^{nd} generation Intel Optane PM with the 3^{rd} generation Intel Xeon Scalable Processors. eADR ensures that data stored within the CPU caches will be flushed to PM upon the power failure. Thus, in eADR-enabled PM systems, the globally visible data is considered persistent, and explicit data flushes are no longer necessary. The emergence of eADR presents unique opportunities to build lock-free data structures and unleash the full potential of PM.

In this paper, we propose NBTree, a lock-free PM-friendly B$^+$-Tree, to deliver high scalability and low PM overhead. To our knowledge, NBTree is the first persistent index designed for eADR-enabled PM systems. To achieve lock-free, NBTree uses atomic primitives to serialize leaf node operations. Moreover, NBTree proposes four novel techniques to enable lock-free access to the leaf during structural modification operations (SMO), including *three-phase SMO*, *sync-on-write*, *sync-on-read*, and *cooperative SMO*. For inner node operations, we develop a *shift-aware search* algorithm to resolve read-write conflicts. To reduce PM overhead, NBTree decouples the leaf nodes into a metadata layer and a key-value layer. The metadata layer is stored in DRAM, along with the inner nodes, to reduce PM accesses. NBTree also adopts *log-structured insert* and *in-place update/delete* to improve cache utilization. Our evaluation shows that NBTree achieves up to 11× higher throughput and 43× lower 99% tail latency than state-of-the-art persistent B$^+$-Trees under YCSB workloads.

PVLDB Reference Format:
Bowen Zhang, Shengan Zheng, Zhenlin Qi, Linpeng Huang. NBTree: a Lock-free PM-friendly Persistent B$^+$-Tree for eADR-enabled PM Systems. PVLDB, 15(6): 1187-1200, 2022.
doi:10.14778/3514061.3514066

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/SJTU-DDST/NBTree.

Proceedings of the VLDB Endowment, Vol. 15, No. 6 ISSN 2150-8097.
doi:10.14778/3514061.3514066
* Linpeng Huang and Shengan Zheng are corresponding authors.

1 INTRODUCTION

Byte-addressable persistent memory (PM), such as Intel Optane DC persistent memory module (DCPMM) [17] is now commercially available. PM offers DRAM-comparable performance as well as disk-like durability. In general, PM-equipped platforms support the asynchronous DRAM refresh (ADR) feature [21], which ensures that the content of the PM DIMMs, as well as the writes that have reached the memory controller's write pending queues (WPQ), survives power failures. However, writes within CPU caches remain volatile. Thus, explicit cache line flush instructions and memory barriers are required to guarantee the persistence of PM writes.

Recently, a new feature called extended ADR (eADR) is available with the arrival of the 3^{rd} generation Intel Xeon Scalable Processors and the 2^{nd} generation Intel Optane DCPMM [24]. Compared with ADR, eADR further guarantees that data within CPU caches will be flushed back to PM after a crash through the reserved energy. It ensures the persistence of the globally visible data in CPU caches and eliminates the need to issue costly synchronous flushes. The emergence of eADR not only facilitates the design of lock-free data structures but reduces PM write overhead.

Building efficient index structures in PM is promising to offer both high performance and data durability for in-memory databases. Most existing persistent indexes [2, 4–6, 28, 32, 37, 40, 43, 47, 59, 61] are solely designed for ADR-based PM systems. On eADR-enabled platforms, an intuitive transformation approach is to simply remove all cache line flush instructions [45]. However, this naïve approach cannot fully exploit the potential of eADR. Those indexes still suffer from two major drawbacks even with the eADR support.

First, existing persistent indexes suffer from inefficient concurrency control. Locks are widely used in persistent indexes because none of the existing primitives can atomically modify and persist data on ADR-based platforms. Atomic CPU hardware primitives, such as Compare-And-Swap (`CAS`), can atomically modify the data but do not guarantee its persistence because CPU caches are volatile. Therefore, without locking, it's possible that a `store` hasn't been persisted before a dependent read from another thread, leading to *dirty read* anomaly. Fortunately, eADR closes the gap between the visibility and persistency of the data in CPU caches, making sure that threads always read persistent data. Thus, eADR provides us an opportunity to develop efficient lock-free data structures.

Second, existing persistent indexes still impose high overheads on PM accesses. Prior researches strive to lower PM overhead by

reducing the number of flush instructions, because data flushing is the primary bottleneck of ADR-based PM systems [37]. With eADR, explicit data flushes to PM are no longer necessary. However, the performance of persistent indexes is still restricted by excessive PM accesses since PM has higher read latency and lower bandwidth than DRAM [16, 52, 55]. Especially for the write operations, although data flushing to PM is off the critical path, dirty cache lines will eventually be written back to PM due to the limited CPU cache capacity. Therefore, it's necessary to redesign the PM-friendly persistent indexes for eADR-enabled PM systems.

In this paper, we present NBTree, a lock-free PM-friendly B^+-Tree to deliver high scalability and low PM overhead. To our knowledge, NBTree is the first PM index based on eADR-enabled PM systems.

To achieve high scalability, NBTree proposes a fully lock-free concurrency control protocol. For leaf node operations, NBTree adopts *log-structured insert* and *in-place update/delete*, combining with `CAS` primitives, to support lock-free accesses. When the inserted leaf is full, NBTree replaces the old leaf with new leaves to maintain the balance of nodes via structural modification operations (SMO). NBTree proposes three novel techniques (*three-phase SMO*, *sync-on-write*, and *sync-on-read*) to deal with the potential anomalies during the lock-free accesses to the leaf in SMO: (1) *Lost update* caused by concurrent updates and deletions to the leaf. NBTree addresses this anomaly by utilizing *three-phase SMO* and *sync-on-write*. When an update or deletion operates on the leaf during SMO, it first in-place modifies the old leaf. Then, the modification is either passively migrated to the new leaf by *three-phase SMO* or actively synchronized to the new leaf using *sync-on-write*. (2) *Dirty or stale read* caused by concurrent search operations. The lock-free search on the SMO leaf might read uncommitted dirty data or stale data. NBTree uses the *sync-on-read* technique to detect and resolve those anomalies. To further reduce tail latency, we propose *cooperative SMO* to make concurrent insertions to the same SMO leaf work cooperatively. For inner node operations, NBTree applies hardware transactional memory (HTM) [26] to achieve atomic writes. Meanwhile, NBTree designs a *shift-aware search* algorithm to ensure the lock-free inner node search reaches the correct leaf.

To reduce PM overhead, NBTree minimizes PM line accesses and improves cache utilization. For leaf nodes in NBTree, the metadata and key-value pairs are decoupled into two layers. The metadata layer is stored in DRAM along with inner nodes. PM only contains the key-value layer so that the number of PM line reads and writes is minimized. The volatile part of NBTree can be rebuilt with the persistent key-value layer after a crash. Moreover, our proposed *log-structured insert* and *in-place update* improve the possibility of *write combining* and *write hits*, optimizing the cache utilization in eADR-enabled PM systems.

In summary, the contributions of this paper include:

- We provide an in-depth analysis of the benefits of the eADR feature. Then, we propose NBTree, the first persistent index based on eADR-enabled PM systems as far as we know.
- We propose lock-free concurrency control for NBTree to achieve high scalability. Our proposed techniques, such as *three-phase SMO*, *sync-on-write*, *sync-on-read*, *cooperative SMO*, and *shift-aware search*, ensure strong consistency for lock-free operations.
- We propose a two-layer leaf node structure for NBTree, which reduces the number of PM line reads and writes in each operation and improves cache utilization.
- We implement NBTree and our evaluation results show that NBTree achieves up to 11× higher throughput and 43× lower 99% tail latency than state-of-the-art counterparts under YCSB workloads.

2 BACKGROUND AND MOTIVATION

In this section, we introduce the background of persistent memory and eADR (Section 2.1), the PM overhead analysis in eADR-enabled PM systems (Section 2.2), and the challenges of designing lock-free persistent data structures (Section 2.3).

2.1 Persistent Memory and eADR

Persistent memory (PM), which is now commercially available, provides many attractive features, such as byte-addressability and data persistency. However, PM still has higher latency and lower bandwidth than DRAM. To reduce the write latency, existing PM-based systems utilize the ADR mechanism [21] to drain the writes sitting on the write pending queues (WPQ) to PM by the reserved energy during a power outage. Therefore, data that reaches the WPQ in ADR-based PM systems is considered persistent, whereas data in the CPU caches remains volatile. As a result, an additional pair of the flush instruction (e.g. `clwb`, `clflush`, `clflushopt`) and memory barrier (e.g. `mfence`, `sfence`) is necessary for programmers to guarantee data persistency [45].

Fortunately, eADR is supported on the 2^{nd} generation Intel Optane DCPMM with the 3^{rd} generation Intel Xeon Scalable Processors. eADR-enabled PM systems reserve more energy that enables them to flush data in CPU caches to PM after a power failure, thereby expanding the persistence domain to include CPU caches [46]. eADR offers the following advantages over ADR.

The first one is reducing PM write overhead. With eADR, the synchronous flush instructions are no longer necessary, which reduces PM write overhead in two aspects. (1) Reducing the latency in the critical path. Previously, the flush instructions and memory barriers result in high latency in the critical path [37]. (2) Saving PM write bandwidth. Delaying writes to PM increases *write hits* and *write combining* in CPU caches, which reduces PM writes.

The second one is facilitating the lock-free design. Data structures can atomically modify and persist data with eADR, which facilitates the lock-free design in PM. Most lock-free data structures [1, 18, 36, 41] rely on atomic CPU hardware primitives, such as `CAS`. However, in ADR-based PM systems, those primitives can atomically modify data but cannot ensure their persistence because CPU caches are volatile. Threads are likely to read unpersisted data in CPU caches, resulting in the *dirty read* anomaly. With eADR, the globally visible data in CPU caches is ensured to be persisted. Thus, it is possible to modify and persist data atomically.

2.2 PM Overhead Analysis

The performance gap between PM and DRAM encourages people to design PM-friendly storage systems to reduce I/O overhead. Previous works [7, 8, 12, 31, 42, 49, 53, 54, 58] designed for ADR-based PM systems mostly focused on reducing the costly flush

Table 1: The PM overhead of tree operations. ($a/b/c$ indicates the PM overhead of an individual insert/delete/update. n indicates the number of key-value pairs in the leaf node.)

	Flush	PM line write	PM line read
NVTree [56]	2/2/2	2/2/2	O(n)
WB$^+$Tree [3]	4/3/3	3/2/2	O(log(n))
FPTree [43]	3/1/3	3/1/3	3
RNTree [39]	2/1/2	3/1/3	O(log(n))
BzTree [2]	15/7/10	11/6/7	O(log(n))
FAST&FAIR [20]	O(n)/O(n)/1	O(n)/O(n)/1	O(n)
uTree [4]	2/1/1	2/2/1	2
NBTree	**1/1/1**	**1/1/1**	**1**

instructions. With eADR, flushing is no longer required. However, although the persistence latency of the writes is hidden by CPU caches, dirty cache lines will eventually be evicted to PM according to the cache replacement policy. Excessive PM writes still result in high latency due to the poor PM write bandwidth. Besides, PM also has higher read latency than DRAM. Thus, the unique features of eADR require a rethinking of how to reduce PM overhead.

We conclude the following three design goals to reduce PM overhead. First, reducing the number of *PM line writes* per operation. *PM line writes* indicate the 64-byte aligned PM lines modified in CPU caches. Reducing *PM line writes* per operation can produce less dirty cache lines, saving PM write bandwidth. Second, increasing the possibility of *write combining* and *write hits* in CPU caches. In this way, multiple write operations can write to the same cache line, reducing PM writes. ADR-based PM systems do not benefit much from it because write operations often need to be synchronously flushed. Third, reducing the number of *PM line reads* per operation. The relatively higher read latency of PM is overlooked in the previous works [3, 39, 56]. However, it is non-negligible, especially in read-intensive data structures, such as B$^+$-Tree.

Table 1 lists PM costs of state-of-the-art persistent B$^+$-Trees and NBTree. We notice that the strategies of reducing the number of flushes sometimes result in fewer *PM line writes*. However, they are not equivalent. RNTree [39], for example, applies selective metadata persistence to reduce the number of flushes but cannot avoid the *PM line writes*. We also find that trees that keep the order of leaf nodes or slot arrays produce non-constant *PM line reads* per operation, which incurs non-negligible overhead.

2.3 The Design Challenges of Lock-free Persistent Data Structures

It's non-trivial to design lock-free data structures (LFD) in PM because they not only need to handle subtle race conditions like volatile indexes, but need to make sure that writes are persisted before any dependent read [14, 51, 62]. There are the following two hardware restrictions to keep us away from designing efficient LFDs in PM. (1) The granularity of atomicity in memory `load` and `store` is one word, that is 8 bytes. Atomic CPU hardware primitives used in many LFDs, such as compare-and-swap (`CAS`), can only atomically modify a single word. However, a single operation in non-trivial data structures needs to read and write multiple words. Therefore, LFDs are likely to expose intermediate states to concurrent threads. Moreover, when a thread is performing an operation, data structures might be changed by other threads. Those problems may result in anomalies such as *lost update*, *stale read*, and *inconsistent read*. (2) ADR-based PM systems do not support atomic primitives to modify and persist the data. Updates are first sent to the CPU caches and then persisted using flush instructions and memory barriers. As the globally visible data in CPU caches are volatile, other threads can easily read unpersisted data, resulting in *dirty read* anomaly.

In the following, we use the persistent B$^+$-Tree as an example to specify how anomalies mentioned above happen.

Lost Update/Stale Read. Updates may be lost permanently, and reads might access stale data due to non-atomic state changes. For example, structural modification operations, such as split, are the most complex state change in B$^+$-Trees. During the split, B$^+$-Tree transfers the content of the old node to newly allocated nodes and then replaces the old node with new nodes. As the split cannot be completed atomically, a concurrent update may occur in the old node but be missed in the new nodes. In this situation, new nodes are facing the risk of the *stale read* anomaly since they are stale. Even worse, if the update is not synchronized to new nodes in a proper way, it will be lost permanently, incurring the *lost update* anomaly. Thus, existing B$^+$-Trees often lock the leaf during the modification. BzTree [2] uses the `PMwCAS` [51] to guarantee the atomicity of writes. However, BzTree performs even worse than lock-based B$^+$-Trees due to the high software overhead of `PMwCAS` [35].

Inconsistent Read. Threads might read the inconsistent state of data structures due to non-atomic state change. For example, the shift operation to keep nodes in B$^+$-Tree sorted cannot be completed atomically. During an insertion or deletion, B$^+$-Tree needs to shift array elements by calling a sequence of `load` and `store` instructions. During shifting, the same entry may appear twice in different slots, which is an inconsistent state that can result in the *inconsistent read*. FAST&FAIR [20] proposes a lock-free search algorithm, which tolerates such inconsistency. During searching, the key is ignored if its left and right child pointers have the same address. However, *inconsistent read* may occur when the state of the node changes between two load operations [57].

Dirty Read. Reads might access the uncommitted dirty data due to non-durable writes. Because of the lack of atomic instructions with the functionality of persistence, there is a temporal gap between when an update becomes globally visible and when it becomes durable. During the update, we firstly store the data in CPU caches, then persist it using a flush instruction and a memory barrier. Other concurrent threads may view the new update before it persists. If a power outage occurs between these two steps, the read operation will get the unpersisted dirty data. Previous works propose several approaches to deal with this issue. ROART [40] and P-ART [32] use the `non-temporal store` to prevent the unpersisted data from being globally visible, but this method does not benefit from CPU caching. `Link-and-persist` [10] and `PMwCAS` [51] use the help mechanism, which allows read threads to flush unpersisted data proactively. However, it adds additional software overhead and design complexity. With eADR, the *dirty read* anomaly is less likely to happen as the globally visible data is always persistent.

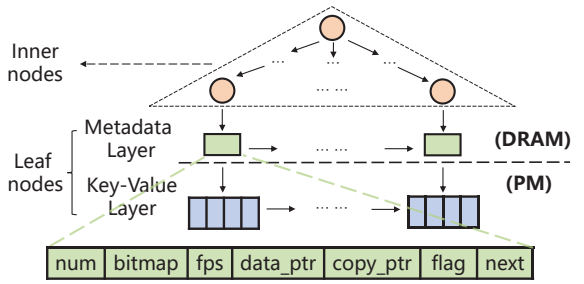


Figure 1: The overall architecture of NBTree.

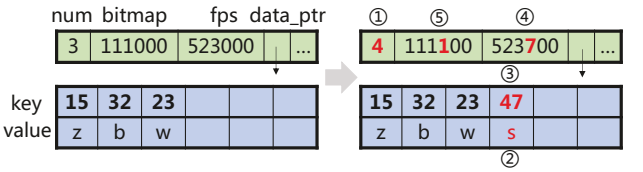


Figure 2: Procedure of an insertion on an NBTree's leaf. (The fingerprints (fps) of the keys are set to $key\%10$ for brevity.)

3 PM-FRIENDLY B$^+$-TREE

NBTree achieves low latency and high scalability by lowering PM overhead in the following two aspects: (1) Reduce the number of *PM line reads/writes* per operation. (2) Leverage the eADR benefits to increase the *write combining* and *write hits* in CPU caches.

In this section, we describe the PM-friendly design of NBTree. We first present the overall architecture (Section 3.1), and then describe the base operations of NBTree (Section 3.2).

3.1 NBTree Structure

The overall architecture of NBTree is shown in Figure 1. In NBTree, the metadata and key-value pairs of the leaf nodes are separated into two layers. The metadata layer, as well as the inner nodes of NBTree, is maintained in DRAM. They can be rebuilt from the persistent key-value layer of leaf nodes in PM during recovery. The two-layer leaf node design enables NBTree to absorb metadata operations in DRAM, reducing PM line accesses drastically.

Specifically, for leaf nodes, both the metadata layer and the key-value layer are linked into a singly linked-list. In the key-value layer, each key-value block is an unsorted array of key-value entries. Each key-value entry stores a 64-bit key and a 64-bit payload. The highest 2 bits (`copy_bit` and `sync_bit`) of the payload are reserved for concurrency control. For variable-sized key-value entries, NBTree stores pointers that indicate the actual keys or values. Each leaf's metadata consists of the following fields: (1) `fps` to store the one-byte fingerprint (hash value) for each key in the leaf, which speeds up key-search on the unsorted array. (2) `num` to store the number of entries occupied by both the committed and in-flight insertions, which handles concurrent insertions. (3) `bitmap` to track the position of the committed insertions in the leaf. (4) `data_ptr` to indicate the address of its key-value block. (5) `copy_ptr` to store the address of newly allocated leaves when the leaf performs SMO. (6) `flag` to track the status of SMO. (7) `next` to indicate the address of the sibling leaf. For inner nodes, NBTree adopts the structure of FAST&FAIR [20], which maintains the sorted array.

3.2 Base operations

NBTree reduces the overhead of base operations (insert, update, delete, and search) by minimizing PM line accesses and maximizing the cache utilization. For each base operation, NBTree first locates the corresponding leaf by searching the inner nodes in DRAM. Then, it uses *log-structured insert*, *in-place update/delete*, and *efficient search* to reduce the average number of *PM line read/writes* on the persistent leaf node to 1 and increase the *write hits* and *write combining*. During recovery, NBTree retrieves all the key-value entries with non-zero keys from the persistent leaf nodes.

Log-structured Insert. Insertions perform in a log-structured manner in NBTree. Figure 2 illustrates the steps of inserting a new key-value pair in NBTree's leaf. First, NBTree increases the `number` (①) to occupy the next free slot. Then, NBTree writes the value (②) and the key (③) to the occupied slot. After writing the key, the new insertion can survive a power failure. Finally, NBTree updates the `fps` (④) and `bitmap` (⑤) to make insertion visible. We observe that the only PM overhead in an insertion is storing a key-value pair. Moreover, with eADR, the *log-structured insert* manner also allows the consecutive insertions on the same leaf to combine in CPU caches, reducing PM writes.

In-place Update/Delete. Conventional log-structured B$^+$-Trees, such as NVTree [56] and RNTree [39], update or delete key-value entries by appending new entries. NBTree, on the other hand, performs *in-place update/delete*. To update a key-value entry, NBTree modifies its value in-place. To delete an entry, NBTree invalidates it by resetting its key to 0. Update and delete can survive system crashes by modifying and persisting the 8-byte key or value with eADR support. In-place update manner is not favored in ADR-based PM systems, as repeatable flushes to the same cache line cause extra latency especially running on skewed workload [5]. With eADR, *in-place* update manner fully utilizes CPU caches and minimizes *PM line writes*.

Efficient Search. The search range is confined to the valid entries indicated by the `bitmap`. This ensures that the entry found by the search operation is persistent and committed. NBTree further narrows the average number of candidate entries to one by checking the `fingerprints`. Finally, NBTree scans the candidate entries to filter the unmatched keys and the deleted keys. In most cases, the search operation produces only one *PM line read*, since the candidate entry is often unique.

Crash Consistency. NBTree can restore its metadata layer and inner nodes using the key-value layer after a crash. During recovery, NBTree scans the list of key-value blocks and labels the slots with non-zero keys as the valid key-value entries. Then, NBTree rebuilds the metadata layer and the inner nodes based on those valid entries. We find that rebuilding NBTree from persistent leaf nodes with 16 million key-value entries takes only 0.32s with a single thread, which is 32× quicker than recovering from the base data.

NBTree maintains consistency even if a crash occurs in the middle of a write operation (insertion/update/deletion). As shown in

Figure 2, during an insertion, writing the key (③) happens after writing the value (②). If the crash happens after writing the key, the intact key-value pair will survive. Otherwise, the in-flight insertion will not leave NBTree in an inconsistent state because the key of the occupied slot is still 0, which is discarded after the recovery. For updates and deletions, they can be completed atomically.

4 LOCK-FREE DESIGN

In this section, we introduce the lock-free concurrency control of NBTree, which is based on a precondition guaranteed by the eADR-enabled platform: **globally visible data is persistent**. We propose different concurrency-control protocols for operations on normal leaf nodes (Section 4.1), leaf nodes during SMO (Section 4.2), and inner nodes (Section 4.3).

4.1 Leaf Node Operations

We divide the base operations in NBTree into two categories. The first category is `insert`, which appends new data to the free slot. The second category is `UDS` operations, including update, deletion, and search. The `UDS` operations always work on the committed insertions. In the following, we discuss how NBTree resolves the `insert-insert`, `UDS-UDS`, and `insert-UDS` conflicts.

Insert-Insert Conflicts. We use atomic primitives to serialize concurrent insertions. As shown in Figure 2, to begin an insertion, NBTree uses the `fetch_and_add` to atomically increase the `num`, occupying the next free slot. This ensures that concurrent insertions are placed in separate slots. At the end of an insertion, NBTree uses `CAS` to atomically update the `bitmap`, which commits the insertion. In this way, NBTree achieves lock-free insert on the leaf nodes.

Insert-UDS Conflicts. Those conflicts are naturally solved in NBTree. Firstly, an insertion always writes the data into the unused space, which does not affect the `UDS` operations. Secondly, NBTree commits an insertion by atomically updating the `bitmap`, which makes the new insertion visible to `UDS` operations. Therefore, `UDS` operations always operate on the completed insertions.

UDS-UDS Conflicts. `UDS-UDS` conflicts in NBTree are resolved in eADR-enabled PM systems without additional overhead. As mentioned above, updates and deletions are completed atomically without exposing the intermediate state. With eADR, those modifications are atomically persisted. Thus, the order of commit and visibility for concurrent updates and deletions are always maintained, and the search operation always reads the latest committed data. Moreover, `UDS` operations are never aborted by other threads, which dramatically improves NBTree's scalability under the workloads with high contentions.

4.2 Structural Modification Operations

Structural modification operations (SMOs) are initiated when a key-value entry is inserted into a full leaf. The conventional procedure of SMO is to copy the entries from the full old leaf to the newly allocated leaves, and then replace the old leaf with the new leaves. However, since the copy phase cannot be completed atomically, lock-free concurrent modifications to the old leaf may not be synchronized to the new leaves, resulting in the *lost update* anomaly.

Table 2: The approaches employed during different phases of SMO to facilitate lock-free leaf node operations.

SMO Phase	Copy	Sync	Link
Update/ Delete	*three-phase SMO* (sync phase)	*sync-on-write*	
Search	unnecessary	*sync-on-read*	unnecessary
Insert	*cooperative SMO*		

Moreover, the lock-free search might read *dirty* or *stale* data due to the inconsistency between the old leaf and new leaves.

Table 2 shows the approaches used by NBTree to resolve the potential anomalies and facilitate lock-free accesses. In NBTree, SMO is divided into three phases (copy phase, sync phase, and link phase). During each phase, different approaches are used to handle concurrent operations on the SMO leaf. For `UD` (update/delete) operations, NBTree resolves the *lost update* anomaly with the sync phase of the SMO and the *sync-on-write* technique. For search operations, NBTree uses *sync-on-read* to prevent the *dirty read* and *stale read* anomaly. We also propose *cooperative SMO*, which enables concurrent insertions to complete SMO cooperatively.

Three-phase SMO. Different from the SMO of traditional B^+-Trees that only includes the copy phase and link phase, NBTree adds a sync phase to avoid the *lost update* anomaly caused by `UD` operations during the copy phase. In the following, we will describe the procedure of each phase.

In the copy phase, SMO copies the valid entries with non-zero keys in the full leaf to new leaves. As shown in Figure 3, NBTree allocates two new leaves if the number of valid entries exceeds a certain threshold (half of the leaf capacity by default). Otherwise, only one new leaf is allocated. Then, NBTree distributes key-value pairs to the new leaves and constructs their metadata layer. Finally, NBTree sets the `copy_ptr` in the old leaf to indicate the address of the first new leaf.

In the sync phase, NBTree synchronizes the lost `UD` operations to new leaves. During the copy phase, concurrent `UD` operations still write to the old leaf. As the copy phase cannot be completed within an atomic instruction, those `UD` operations, such as `Update(2,s)` in Figure 3, might not have been migrated to new leaves yet. Therefore, in the sync phase, NBTree employs `CAS` to synchronize the missed `UD` operations to new leaves.

The link phase replaces the old leaf in NBTree with new leaves. NBTree firstly links new leaves into the singly linked-list of the key-value layer and the metadata layer by changing the `next` pointer of the previous leaf. Then, NBTree installs new leaves to the parent node (described in Section 4.3).

Sync-on-write. In the post-copy phases of SMO, `UD` operations resolve the *lost update* anomaly by adopting a *sync-on-write* approach, which actively synchronizes the modification from the old leaf to the new leaf. Specifically, for an update, after modifying a key-value in the old leaf, it re-searches the target key in the new leaf. If the corresponding value in the new leaf is not up-to-date, NBTree synchronizes the latest update to the new leaf using `CAS`.

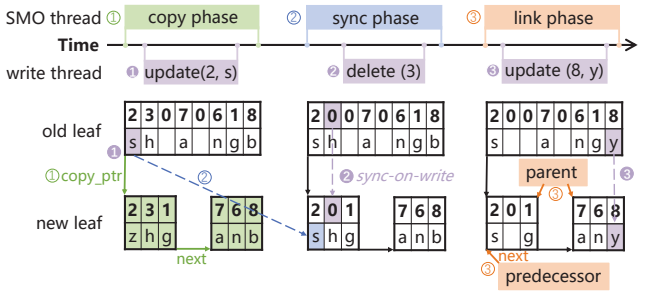


Figure 3: The procedure of *three-phase SMO* and *sync-on-write* when updates and deletions operate on an SMO leaf.

With the support of eADR, CAS can atomically modify and persist the synchronization. Similar to the update, the deletion also re-executes in the new leaf if it contains the target key. NBTree imposes low-overhead on *sync-on-write* because it only incurs one additional search on the new leaf and one CAS primitive.

During the post-copy phases of SMO, *sync-on-write* prevents lock-free UD operations from suffering the *lost update* anomaly. As illustrated in Figure 3, during the copy phase, any UD operation that happens on the old leaf (e.g. `update(2,s)`) will be synchronized to the new leaf in SMO's sync phase. However, UD operations happen after the copy phase (e.g. `update(8,y)`, `delete(3)`) may still be lost. To avoid the *lost update* anomaly, UD operations need to actively synchronize the modification by calling *sync-on-write*.

Through *sync-on-write* and *three-phase SMO*, we ensure that NBTree always maintains a consistent state that includes all committed operations after a crash. As we previously mentioned, SMO's *durability point* is when the new leaves replace the old leaf in PM by linking themselves into the key-value layer during the link phase. If a crash occurs before the *durability point*, the old leaf will remain in the key-value layer. During SMO, any modification must first operate on the old leaf. Therefore, as illustrated in Table 3, the old leaf always holds both the *latest* committed and uncommitted operations during SMO. The uncommitted operations in the old leaf won't cause inconsistency because the in-flight *sync-on-write* is unnecessary after discarding new leaves. If a crash occurs after the *durability point*, the new leaf will be linked into the key-value layer. At that time, all UD operations committed in the copy phase have been synchronized to the new leaf in the sync phase. For UD operations that happen after the copy phase, they are only committed when they write to a new leaf. As a result, new leaves hold the *latest* committed operations after the *durability point*. Besides, with the support of eADR, the new leaf is consistent as the *sync-on-write* is atomic. To summarize, the consistent leaf with the *latest* committed operations will survive whenever the crash happens.

SMO threads (sync phase) and UD threads (*sync-on-write*) may synchronize the same value to the new leaf concurrently. NBTree can serialize those synchronizations using the highest two bits of each entry's value. We will discuss this scenario in Section 5.

Sync-on-read. To deal with the potential inconsistency between the old leaf and the new leaves, the search operations employ the *sync-on-read* approach to synchronize the corresponding key-value entries from the old leaf to the new leaf. Specifically, NBTree searches the target key in both old and new leaves. If the returned results differ, the search operation updates or deletes the key-value in the new leaf to match the one in the old leaf. The overhead of the *sync-on-read* is as low as the *sync-on-write*.

Table 3: The latest and clean leaves in different phases of SMO. (The *latest* leaf contains all committed writes. The *Clean* leaf does not contain any uncommitted dirty write.)

SMO Phase	Latest		Clean	
	old leaf	new leaf	old leaf	new leaf
Copy	✓		✓	
Sync	✓			✓
Link	✓	✓		✓
After SMO		✓		✓

Sync-on-read guarantees that a lock-free concurrent search returns the latest and committed version of a key-value entry. During SMO's sync phase, reading from either old or new leaves without performing *sync-on-read* may lead to the *stale read* or *dirty read* anomaly. As illustrated in Table 3, in the sync phase, the old leaf is possibly *dirty* because the UD operations might have not committed due to an on-going *sync-on-write*. Meanwhile, the new leaf is likely *stale* as the SMO thread may not have finished synchronizing the *latest* modification that happened during the copy phase. To address this problem, the search operation uses *sync-on-read* to synchronize the *latest* key-value from the old leaf to the new leaf. It makes sure that the target key-value pair in the new leaf is both the *latest* and *clean* before returning the search result.

Search operations on the leaf that is not in the sync phase can directly read the correct value without calling *sync-on-read*. Table 3 shows the destination of reads, which is the leaf that holds both the *latest* and *clean* key-value pairs. During the copy phase, incoming reads go to the old leaf, which is both the *latest* and *clean* since concurrent UD operations directly commit in the copy phase. After the sync phase, reads go to the new leaf. This is because previous UD operations that happened in the copy phase have already been synchronized to the new leaf, while later UD operations are committed once they are visible in the new leaf with the eADR support.

Cooperative SMO. In NBTree, concurrent insertions to the leaf during SMO employ *cooperative SMO*. The insertion thread that encounters a leaf with an in-flight SMO will help complete its SMO before continuing. NBTree uses atomic primitives, such as CAS, to coordinate multiple SMO threads, making sure only the fastest modification can be visible. In this way, instead of waiting for the completion of SMO, NBTree guarantees that SMO moves forward at the fastest speed, even when a certain SMO thread is suspended.

Specifically, in the copy phase, multiple SMO threads prepare new nodes respectively and use CAS primitive to atomically install the `copy_ptr`. In the sync phase, the synchronization of each key-value entry can also be completed cooperatively by using CAS primitive. In the link phase, NBTree uses CAS to link the new leaf into the metadata layer and the key-value layer. Then, NBTree uses HTM (described in Section 4.3) to atomically update the parent node and set `flag.link`.

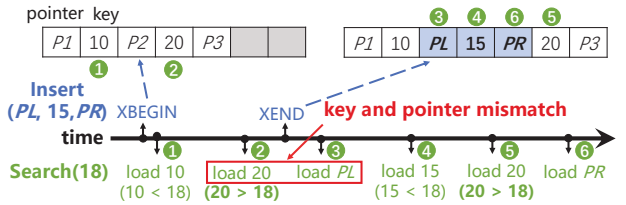


Figure 4: *Shift-aware search* on the inner node under the read-write conflict. (XBEGIN means the start of the transaction, XEND means the end of the transaction.)

Algorithm 1: Insert(K key, V val)

```
1 leaf = findLeaf(key);
2 pos = fetch_and_add(&leaf→num, 1);
3 if pos >= LEAF_NODE_SIZE then
4 |   leaf→setSMOBit();    // Set the highest bit of bitmap
5 |   SMO(leaf);
6 |   goto Line 1;
7 leaf→insert(key, val, pos);
8 if ! leaf→atomicSetBitmap(pos) then
9 |   goto Line 1;
```

4.3 Inner Node Operations

We propose a *shift-aware search* algorithm and use the *HTM-based update* to coordinate concurrent inner node operations.

HTM-based Update. NBTree uses HTM to atomically update inner nodes following FPTree [43]. HTM is an optimistic concurrency control tool that uses hardware transactions to make multiple writes atomically visible. It is non-blocking since it only aborts when conflicts are detected. Wrapping updates in HTM is efficient because the conflict of inner node modifications rarely happens. Moreover, updates do not expose intermediate states to other threads, which avoids the read thread viewing the inconsistent state.

Shift-aware Search. Although the modifications to the inner nodes are atomic, lock-free inner node search might find the wrong child pointer due to the read-write conflict. As illustrated in Figure 4, the search operation performs a linear search (`Search(18)`) to retrieve the key of 20 (②). However, before it fetches the corresponding pointer *P2*, an insertion (`Insert(PL,15,PR)`) modifies the node. Therefore, the search operation loads the wrong pointer *PL* (③) instead of *P2* or *PR*. As a result, an existing key might be missed in the search operations.

NBTree proposes a *shift-aware search* algorithm to ensure the correctness of lock-free inner node search despite concurrent write transactions (insertion or deletion) on the same node. As shown in Figure 4, before proceeding to the next level of the tree through *PL*, NBTree checks if the fetched key (20 in ②) has been shifted by concurrent write transactions. If so, NBTree re-searches the node from the current position (④-⑥), making sure that the target key lies within the sub-tree indicated by the returned pointer.

The *shift-aware search* algorithm always finds the correct pointer for the following two reasons. First, NBTree keeps searching and data shifting in the same direction. As shown in Figure 4, during the search, the insertion shifts the data from left to right. If the search proceeds in the same direction, then it never misses the newly inserted key. Inspired by FAST&FAIR, we maintain a `switch_counter` in each node, which is increased when the insertion and deletion on the inner node take turns. Second, we adopt the concurrency protocol of the B-link tree [33] to handle SMO. NBTree maintains a `high_key` (the largest key in the node) and `sibling_ptr` in each inner node. NBTree re-searches in the sibling node if the target key is less than the `high_key`, which indicates an SMO has happened on the node.

Shift-aware search is efficient for two reasons: (1) It achieves lock-free search without using HTM, avoiding transaction abortions and hardware overheads. (2) Linear search on small arrays is more efficient than binary search for its better cache locality, which is illustrated in FAST&FAIR [20].

5 IMPLEMENTATION

In this section, we first describe the implementation of NBTree: insert (Section 5.1), update/delete (Section 5.2), and search (Section 5.3). Then we present the limitation of our work (Section 5.4).

5.1 Insert

Algorithm 1 describes the insert operation. After locating the target leaf, NBTree occupies the next free slot using the atomic primitive (Line 2). If the leaf is full, NBTree initiates SMO by setting the `smo bit`, the highest bit of the `bitmap` (Line 3-5). NBTree commits the insertion by updating the `bitmap` using `CAS` (Line 8).

The procedure of SMO is listed in Algorithm 2. In the copy phase (Line 2-9), NBTree distributes valid entries with non-zero keys to newly allocated leaf nodes and constructs the metadata layers for the new leaves (Line 4-8). For each copied entry, NBTree sets the highest bit (`copy_bit`) of the value (Line 5). The `copy_bit` indicates that the value is copied from the old leaf in the copy phase.

In the sync phase (Line 11-24), NBTree sequentially obtains the valid entries in the old leaf (`entry`, Line 11) and new leaves (`syncEntry`, Line 13). If their keys match but values mismatch (Line 14-16), NBTree uses `CAS` to synchronize the lost update, clear the `copy_bit` and set the `sync_bit` (the second highest bit) of the target value in the new leaf. The `sync_bit` indicates the value is synchronized from the old leaf. `CAS` will abort if the `copy_bit` of the target value has been cleared. This ensures that any key-value pair is synchronized at most once during the sync phase. If the key of `entry` and `syncEntry` mismatch, NBTree synchronizes the lost deletion because it usually indicates the key of `syncEntry` has been deleted in the old leaf (Line 19-22). However, there are two exceptions that the key of `entry` has been deleted in the new leaf. (1) SMO has been completed by another SMO thread. As a result, the latest operations directly delete the entry in the new leaf without having to go through the old leaf. In this case, NBTree directly aborts SMO (Line 17). (2) A concurrent operation deletes the key of `entry` on the old leaf after the `entry` has been read. Meanwhile, the deletion has been synchronized by other threads before `syncEntry` is read. In this case, synchronization is not required (Line 18).

In the link phase (Line 26-36), NBTree uses `CAS` to link the leaf to both the key-value layer and the metadata layer (Line 27-28). If the previous leaf is in SMO, NBTree will join the *cooperative SMO* to

Algorithm 2: SMO(Leaf leaf)

```
1  if ! leaf→copy_ptr then // Copy phase
2      splitKey = leaf→findSplitKey();
3      leftLeaf, rightLeaf = allocNewNode();
4      while (entry = leaf→next()).key != 0 do
5          entry.val |= copy_bit;              // Set the copy_bit
6          copy(entry, leftLeaf, rightLeaf, splitKey)
7      end
8      setMetadata(leftLeaf, rightLeaf);
9      CAS(&(leaf→copy_ptr), NULL, leftLeaf);
10 if ! leaf→flag.sync then // Sync phase
11     while (entry = (key, val) = leaf→next()).key!=0 do
12         syncLeaf = key < splitKey ? leftLeaf : rightLeaf;
13         (k, v) = syncEntry = syncLeaf→next();
14         if k == key then
15             if (v & ~(copy_bit | sync_bit)) != (val &= ~(copy_bit |
                sync_bit)) then
16                 CAS(&syncEntry.val, v|copy_bit, val|sync_bit);
17         else if leaf→flag.link then return ;
18         else if ! entry.key then syncLeaf→prev() ;
19         else
20             syncEntry.key = 0;
21             mfence();
22             goto Line 13;
23     end
24     leaf→flag.sync = 1;
25 if ! leaf→flag.link then // Link phase
26     pred = findPredLeaf(key);
27     CAS(&(pred→data_ptr→next), leaf→data_ptr,
        leftLeaf→data_ptr);
28     CAS(&(pred→next), leaf, leftLeaf);
29     if pred→isSMO() then
30         SMO(pred);
31         goto Line 25;
32     xbegin();                          // Start an HTM transaction
33     if ! leaf→flag.link then
34         update_parent(leaf);
35         leaf→flag.link = 1;
36     xend();
```

Algorithm 3: Update(K key, V val)

```
1  leaf = findLeaf(key);
2  entry = leaf→update(key, val);
3  if leaf→isSMO() and leaf→copy_ptr then
4      (nKey, nVal) = nEntry = leaf→copy_ptr→find(key);
5      if nKey and ((v=nVal) & ~(copy_bit | sync_bit)) !=
        (val=entry.val) and v & (copy_bit | sync_bit) then
6          if ! CAS(&nVal, v, val|sync_bit) then
7              goto Line 5;
8  return true;
```

Algorithm 4: Delete(K key)

```
1  leaf = findLeaf(key);
2  leaf→delete(key);
3  if leaf→isSMO() and leaf→copy_ptr then
4      nEntry = leaf→copy_ptr→find(key);
5      if nEntry then
6          nEntry.key = 0;
7          mfence();
8  return true;
```

Algorithm 5: Search(K key)

```
1  leaf = findLeaf(key);
2  entry = leaf→find(key);
3  val = entry.key ? (entry.val & ~(copy_bit | sync_bit)) : 0;
4  if leaf→isSMO() and leaf→copy_ptr then
5      if leaf→flag.sync then return leaf→copy_ptr→search(key) ;
6      (nKey, nVal) = nEntry = leaf→copy_ptr→find(key);
7      if ! nKey then return 0 ;
8      if entry.key == 0 then
9          nEntry.key = 0;
10         mfence();
11     else if (nVal & ~(copy_bit | sync_bit)) != val then
12         CAS(&(nEntry.val), nVal|copy_bit, val|sync_bit);
13     val = nEntry.key ? (nEntry.val & ~(copy_bit | sync_bit)) : 0;
14 return val;
```

avoid the lost update of `next` pointer (Line 29-31). Finally, NBTree employs HTM to update the parent node and set the `flag.link` atomically (Line 32-36).

NBTree adopts the epoch-based garbage collection [13] to reclaim the old leaves. Epoch-based garbage collection recycles the old leaves two epochs after the end of SMO, which ensures that those leaves have no concurrent references.

5.2 Update/Delete

Algorithm 3 shows the process of the update operation. NBTree firstly performs an *in-place update* in the target leaf (Line 1-2). If the leaf is in the post-copy phases of its SMO and the target key exists in the new leaf but its value is not the latest, NBTree will perform *sync-on-write* via `CAS` (Line 5-6). *Sync-on-write* clears the `copy_bit`, which prevents the synchronization invoked by SMO threads from overwriting the current update. It also sets the `sync_bit` to distinguish itself from the update directly operated on the new leaf. When an update performs *sync-on-write*, the SMO might have been completed by other threads. At that time, the latest updates directly operate on the new leaf without setting `sync_bit` or `copy_bit`. *Sync-on-write* does not overwrite those new updates to keep the linearizability. Besides, if the value in the old leaf has been changed by new updates, NBTree will synchronize the latest one.

Algorithm 4 depicts the delete operation. Similar to the update, it will synchronize the deletion if necessary.

5.3 Search

The search operation is shown in Algorithm 5. For the inner node search (Line 1), we directly reuse the code of FAST&FAIR and add the key-checking procedure before returning the child pointer to detect if any update happens. For the leaf node search (Line 2-14), NBTree directly returns the search result on the target leaf if SMO

is not taking place or it is in the copy phase. Otherwise, NBTree searches the target key in the new leaf (Line 4-13). NBTree performs *sync-on-read* if the SMO is in the sync phase and the search results in two leaves mismatch (Line 8-13).

5.4 Limitation

In the current design, NBTree does not address the NUMA-related performance issues in PM. Similar to the prior work on persistent indexes [4, 20, 42], NBTree cannot scale well across multiple NUMA nodes for the following two reasons. First, due to the multi-socket cache coherence traffic, accessing PM on a remote NUMA node has much lower bandwidth than accessing local PM, according to recent studies [25, 27, 50]. Second, atomic primitives used in NBTree perform poorly across NUMA nodes. We plan to address the scalability issues of NBTree in the environment of multiple NUMA nodes in our future work.

6 EVALUATION

In this section, we evaluate the performance of NBTree against other state-of-the-art persistent B$^+$-Trees. We first describe our experiment setup (Section 6.1). Then, we perform single-threaded evaluation (Section 6.2), multi-threaded evaluation (Section 6.3), and YCSB evaluation (Section 6.4). After that, we compare the performance of indexes in two persistence modes (Section 6.5). Finally, we evaluate the performance in real-world systems (Section 6.6).

6.1 Experiment Setup

Testbed. Our testbed machine is a dual-socket Dell R750 server with two Intel Xeon Gold 6348 CPUs, the third generation Xeon Scalable processors that support eADR and TSX. Each CPU has 28 cores and a shared 42MB L3 cache, while each CPU core has a 48KB L1D cache, 32KB L1I cache, 1280KB L2 cache. The system is equipped with 512GB DRAM and 4TB PM (eight 256GB Barlow Pass DIMM per socket). In our evaluation, threads are pinned to NUMA node 0, and are only allowed to access the local DRAM and PM to avoid NUMA effects. We install a PM-aware file system (Ext4-DAX) in `fsdax` mode to manage PM devices. Then, we map large files into the virtual address using PMDK [23] to serve tree nodes allocation. We evaluate the performance of two persistence modes, eADR and ADR. To persist a `store`, we use `clwb` and `mfence` in ADR mode and solely use `mfence` in eADR mode.

Compared Systems. We compare NBTree against seven state-of-the-art persistent B$^+$-Trees, including NVTree, WB$^+$Tree, FPTree, RNTree, BzTree, FAST&FAIR, and uTree. We directly use the open-sourced code of uTree [15], FAST&FAIR [29], BzTree [30], and RNTree [38]. We borrow Liu's [38] implementations of WB$^+$Tree, FPTree, and NVTree. We skip the evaluation of the multi-threaded performance of NVTree and WB$^+$Tree as their implementations do not support concurrency control. For variable-sized keys, we only compare NBTree with BzTree as the implementations of other trees do not support this function.

Default Configuration. We warm up each tree with 16 million key-value pairs and then run enough time for different workloads. By default, we use 8-byte keys and values. For variable-sized keys and values, we store them in the external memory region, and only keep the pointers (48-bit) in indexes to indicate their addresses. The node size of each tree is configured to 1KB. We run all trees in eADR mode except in Section 6.5.

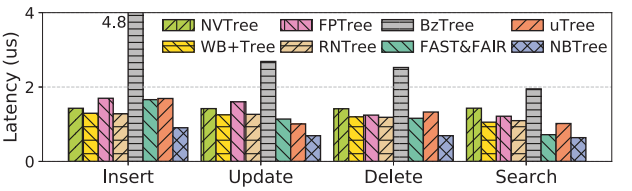


Figure 5: The average latency of base operations. (Single thread, uniform access.)

6.2 Single Thread Evaluation

In this section, we evaluate the single-thread performance of base operations (search, insert, update, and delete) in eADR mode. We run individual operations under random key-access distribution and then calculate the average latency.

As shown in Figure 5, NBTree achieves the lowest latency in every base operation. As the persistence overhead of a write is hidden by CPU caches in eADR mode, we attribute the good performance of NBTree to low *PM line reads*. In most cases, NBTree only causes one *PM line read* in each operation because it places the metadata of the leaf nodes in DRAM and uses fingerprints to filter the unmatched keys. The only PM overhead of NBTree comes from accessing the matched key-value pairs.

In contrast, as illustrated in Table 1, other persistent B$^+$-Trees produce more *PM line reads*, resulting in higher latency. We conclude the sources of *PM line reads* in the following: (1) Most of the B$^+$-Trees need to access the metadata of the leaf node. The metadata is often stored in different PM lines from the actual key-value pair, resulting in additional *PM line reads*. (2) Searching in the leaf node causes multiple *PM line reads*. FAST&FAIR and NVTree use linear search to locate the key-value pair, which needs to traverse half of the leaf on average. WB$^+$Tree, RNTree, and BzTree perform the binary search, which has a similar PM overhead to the linear search when the array size is small. (3) FAST&FAIR and BzTree produce extra *PM line reads* when they perform inner node search because they store inner nodes in PM. (4) uTree invokes additional *PM line reads* to access the sibling node in the linked list, which shows poor locality with the current node. (5) BzTree applies `PMwCAS` [51] to atomically persist the modification. Each `PMwCAS` produces multiple *PM line reads* to access the descriptors.

6.3 Multi-threaded Evaluation

We evaluate the multi-threaded performance of base operations under random key access distribution. As shown in Figure 6, NBTree achieves the highest throughput in each operation. Compared with other trees, the throughput of NBTree in 56 threads is 1.6-7.5× higher on insert, 1.5-5.0× higher on update, 2.0-4.9× higher on delete, 1.6-5.1× higher on search. This is primarily because NBTree minimizes both PM line reads and writes. Reducing *PM line writes* in eADR-enabled PM systems is important. The reason is that the modified PM lines in CPU caches are eventually evicted to PM with

Figure 6: The throughput of base operations. (56 threads, uniform access. NBTree-SL applies the single-layer leaf nodes.)

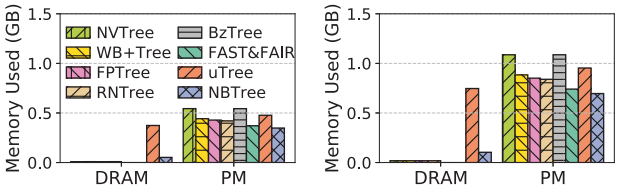


Figure 7: The space consumption of DRAM and PM after initializing the trees with 16M key-value entries (Left) and inserting 16M more key-value entries to the trees (Right).

low bandwidth. Multi-threaded writes can saturate CPU caches and WPQ, resulting in high latency. Besides, excessive *PM line reads* also degrade the multi-threaded performance due to the high latency.

NBTree scales well in the multi-threaded evaluation as it limits the *PM line read/writes* per operation to 1 in most cases. Figure 6 also shows that the two-layered leaf node design speeds up the insert operations by 2.4× because it absorbs metadata modifications in DRAM. Furthermore, although the UDS operations do not modify the metadata due to the optimization of NBTree, the two-layered leaf node design still improves their throughputs by up to 29% due to fewer *PM line reads*. Meanwhile, the additional DRAM consumption from the metadata layer is tolerable. As shown in Figure 7, the ratio of DRAM and PM consumption in NBTree is around 1:7, which is close to the ratio of our testbed configuration (1:8). In practice, this ratio will be much smaller if the value size is large because the values only reside in PM.

As shown in Table 1, other persistent indexes have lower scalability for their high *PM line read/writes*. We have analyzed the cost of PM reads in detail in Section 6.2. Compared with NBTree, other trees produce extra *PM line writes* in the following aspects: (1) They modify the persistent metadata of the leaf nodes for various usages, such as correct recovery, traversal acceleration, and concurrency control. (2) BzTree produces the most *PM line writes* because it needs to record a descriptor in each PMwCAS, resulting in the lowest scalability. (3) FAST&FAIR causes additional *PM line writes* to maintain the order of leaf nodes. As a result, its throughputs on insertions and deletions are low.

6.4 YCSB Evaluation

In this section, we evaluate the performance of persistent B$^+$-Trees with real-world YCSB [9] workloads. We generate the skewed (zipfian key access distribution) and read-write mixture workloads

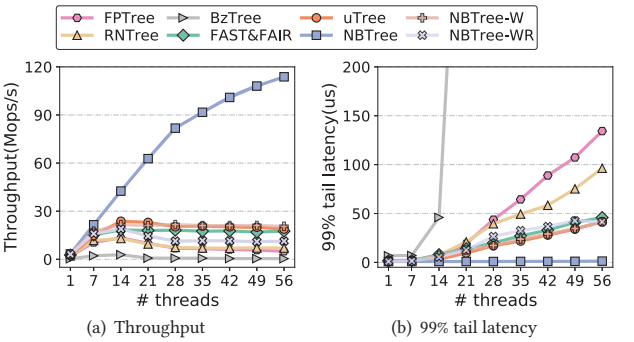


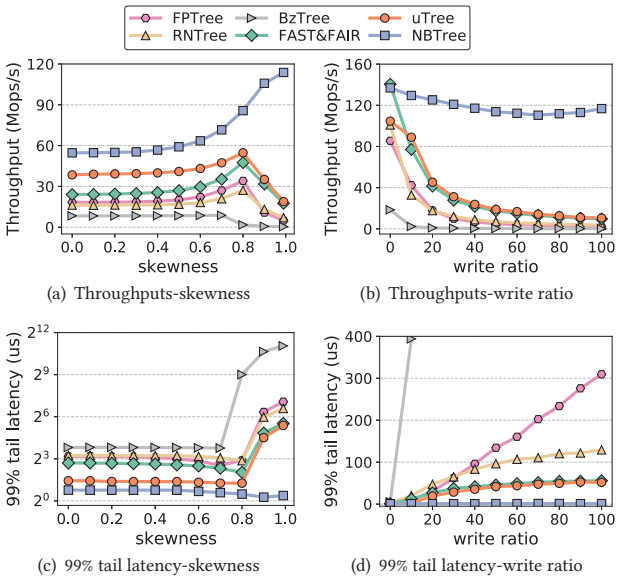
Figure 8: The throughput and 99% tail latency under YCSB workload. (Read:write=50:50, skewness=0.99. NBTree-W disables lock-free write schemes. NBTree-WR disables both lock-free write and read schemes.)

based on YCSB. By default, the write operations in the workload are upsert. Upsert will insert a new key if the target key does not exist. Otherwise, it performs an update.

Overall Evaluation. Figure 8 reports the evaluation results under YCSB workload (read:write=50:50) in a zipfian key access distribution with the default 0.99 skewness. We observe that NBTree has almost linear scalability on throughput and near-constant 99% tail latency with the increase of threads, while other trees only scale up to 14 threads. In 56 threads, NBTree achieves 6.0× higher throughput and 32× lower 99% tail latency than other trees.

We attribute the high performance of NBTree to our efficient lock-free design. The skewed workload often introduces a lot of leaf-level conflicts. The lock-free leaf node operations in NBTree can scale well under high contentions. When operating on the leaf that is not performing SMO, NBTree only employs a small number of atomic primitives to support lock-free access. When operating on the leaves in SMO, UDS operations fix the potential anomaly by our proposed techniques, such as *three-phase SMO*, *sync-on-write*, and *sync-on-read*, which only introduces at most one additional leaf node search and one CAS primitive. Concurrent insertions also apply *cooperative SMO* to achieve lock-free accesses. Besides, as the writes happen infrequently in inner nodes, our proposed *shift-aware search* and *HTM-based* updates can also scale well.

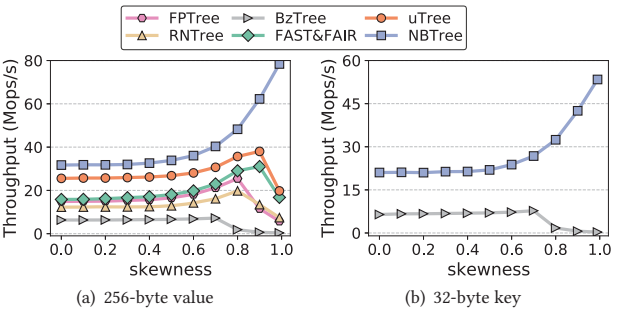
To better demonstrate the impact of our lock-free design and illustrate the performance gap between NBTree and other indexes, we implement two additional versions of NBTree. NBTree-W disables our lock-free write schemes and applies node-grained write-locks instead. As shown in Figure 8, NBTree achieves 5.5× higher throughput than NBTree-W due to our lock-free write design. Meanwhile, FAST&FAIR and uTree achieve similar performance with NBTree-W as they use similar schemes for concurrency control. NBTree-WR further disables our lock-free read approaches and replaces them with HTM-based read, which is used in FPTree and RNTree. We find that NBTree-W achieves 1.8× higher throughput than NBTree-WR. Moreover, NBTree-WR is still 1.9× faster than RNTree and 2.6× faster than FPTree due to our PM-friendly design. As for BzTree, it achieves lock-free by utilizing PMwCAS, which is an optimistic approach implemented by a series of CAS and RDCSS [19]



(a) Throughputs-skewness (b) Throughputs-write ratio

(c) 99% tail latency-skewness (d) 99% tail latency-write ratio

Figure 9: The performance varying the write ratio and skewness under YCSB workload.



(a) 256-byte value (b) 32-byte key

Figure 10: The YCSB performance of the indexes with large key/value. (Read:write=50:50, skewness=0.99.)

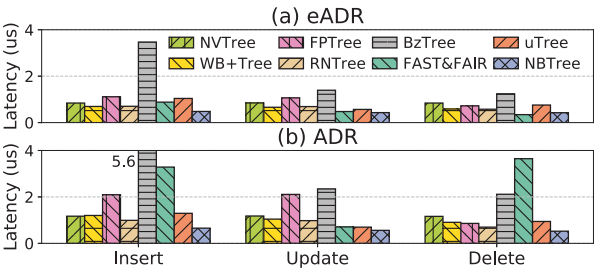


Figure 11: The PM overhead of the leaf nodes in eADR/ADR mode. (Single thread, uniform access.)

operations. However, PMwCAS is vulnerable to high contentions and brings high software overhead [35]. Therefore, BzTree has the worst performance despite its lock-free design.

Effect of Skewness. Figure 9(a) and Figure 9(c) report the evaluation results when we vary the skewness (zipfian coefficient) in YCSB workload (read:write=50:50). We notice that NBTree has better performance with the increase of skewness. The reason is two-fold: (1) Our efficient lock-free designs prevent the concurrency control from becoming the performance bottleneck. (2) Our cache-friendly designs, including *in-place update* and *log-structured insert*, have larger effects with the increase of the skewness. Those designs increase the possibility of *write combining* and *write hits* in CPU caches, which saves the PM write bandwidth.

With the increase of skewness, other trees have a slight performance improvement when the skewness is less than 0.8, which benefits from better cache utilization. When the skewness is larger than 0.8, they have a dramatic performance drop because they cannot scale well under frequent leaf-level contentions.

Effect of Write Ratio. Figure 9(b) and Figure 9(d) show the evaluation results when we vary the write ratio of the YCSB workload (skewness=0.99). We observe that the performance gap becomes larger between NBTree and other B$^+$-Trees with the increase of write ratio. NBTree achieves 11× higher throughput and 43× lower 99% tail latency under the write-only workload. This is because NBTree applies efficient lock-free algorithms for both reads and writes. In contrast, previous works focus on the optimization of concurrent reads but do not support efficient concurrent writes.

Effect of Large Key/Value. Figure 10 reports the evaluation of indexes when the key-value size is larger than 8 bytes. We observe that NBTree still achieves significantly higher throughput than other indexes, especially when the skewness is high. However, the performance gap between NBTree and other indexes becomes smaller. The reason is two-fold: (1) The PM write bandwidth is dominated by persisting large values, dwarfing the performance benefits from our optimization on NBTree. (2) NBTree employs the 8-byte pointers to indicate variable-sized keys, which incurs a lot of pointer dereferences in inner node search. In contrast, Bztree continuously stores the variable-sized keys in a single node, which avoids expensive pointer dereferences.

6.5 Comparison of Persistence Modes

In this section, we first compare the performance of persistent indexes between two persistence modes: ADR and eADR. Then, we evaluate the performance of NBTree in the pure DRAM setting.

Figure 11 reports the PM overhead of the leaf nodes in two persistence modes. Firstly, we find that the PM overhead of all trees is reduced in eADR mode, compared with ADR mode. The primary reason is that the latency on the critical path caused by flush instructions is removed. For example, FAST&FAIR has a significant performance improvement because eADR minimizes the large overhead of data shifts to keep arrays sorted. The performance of BzTree also improves a lot because a large number of flush instructions needed by PMwCAS are removed. However, PMwCAS is still costly due to excessive PM accesses, resulting in the poor performance of BzTree. Secondly, we observe that NBTree has the lowest PM overhead in ADR mode. This is attributed to our PM-friendly designs, which cost only one flush instruction in each write operation.

Figure 12 shows the performance of two persistence modes under YCSB workload (56 threads, read:write=50:50, skewness=0.99). We

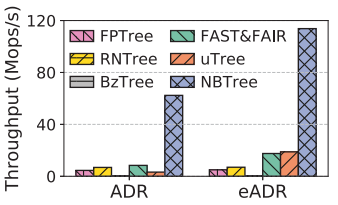


Figure 12: The performance in ADR/eADR mode under YCSB workloads.

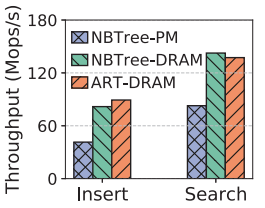


Figure 13: The performance of base operations on the DRAM setting.

have the following four observations. First, NBTree achieves the most performance improvement with the eADR support. This is because the cache-friendly designs (e.g. *in-place update*, and *log-structured insert*) of NBTree take effect in eADR mode. Second, NBTree also performs the best among all trees in ADR mode due to the lock-free design. However, the *dirty read* anomaly is likely to happen in ADR mode because the CPU caches are volatile. Third, uTree and FAST&FAIR also speed up significantly because of the better cache utilization in eADR mode. Fourth, RNTree, FPTree, and BzTree do not benefit from eADR because their concurrency control is vulnerable to high contentions.

Figure 13 shows the performance of NBTree in a pure DRAM setting (uniform access). We observe that NBTree achieves 1.7× higher throughput on search operations and 1.8× higher throughput on insert operations when running on DRAM. We attribute this to the limited bandwidth and high latency of PM compared with DRAM. Meanwhile, NBTree has comparable performance with state-of-the-art indexes (e.g. ART [34]) designed for DRAM because NBTree avoids using redundant operations to guarantee durability.

6.6 End-to-End Evaluation

Redis [44] is a popular in-memory key-value store using a hash table as its index. We use the multi-threaded version of the Redis [48] and replace its internal index with our evaluated trees. We run 28 threads on the Redis server in our evaluation. NBTree achieves the throughput of 1719.4 Kops/s, which is 1.13-1.53× higher than other state-of-the-art persistent B$^+$-Trees under the YCSB-A workload [9]. The evaluation results confirm our previous experiments. Note that the performance gap among indexes becomes smaller due to the high software overhead of Redis.

7 RELATED WORK

Indexes Optimized for PM. In ADR-based PM systems, the slow write is the performance bottleneck of persistent indexes because flush instructions introduce high latency on the critical path. Therefore, previous works have proposed various ways to optimize the write performance of persistent indexes. Most persistent B$^+$-Trees, such as WB$^+$Tree [3], NVTree [56], FPTree [43], RNTree [39], and LB$^+$Tree [37], implement the unsorted leaf nodes. The reason is that inserting or deleting an element of the sorted leaf node produces lots of PM writes to shift array elements. FPTree firstly proposes the selective persistence technique to place inner nodes in DRAM, which speeds up the inner node operations. The volatile inner nodes can be reconstructed from persistent leaf nodes after a crash. RNTree and ROART [40] further remove the flush instructions when modifying reconstructable metadata in the leaf node to reduce the critical path latency. uTree places the sorted leaf nodes in DRAM and adds a persistent shadow list-based layer to ensure crash consistency. In this way, uTree offloads the expensive structural refinement operations (SRO) to DRAM. Persistent hash indexes, such as level hashing [61], path hashing [60], and CCEH [42], also make lots of efforts to write-efficient designs.

Concurrency Control for Persistent Indexes. Previous works propose various concurrency control strategies for persistent indexes to leverage the benefits of multi-core processors. For persistent B$^+$-Trees, FPTree proposes the selective concurrency technique, which handles the concurrency of inner nodes by HTM and serializes the accesses of leaf nodes by the node-grained locks. It improves the scalability in the situation with infrequent contentions but performs poorly in skewed workloads. Based on FPTree, RNTree excludes some slow persistent instructions out of the critical section to achieve more concurrency in the leaf nodes. FAST&FAIR designs a lock-free search algorithm inspired by B-link tree [33], which tolerates the transient inconsistent states caused by write transactions. It improves search performance but tends to cause consistency problem [32]. uTree supports lock-free concurrency control for the list layer but still uses the coarse-grained locks in the leaf nodes. BzTree [2] develops the first lock-free persistent B$^+$-Tree with `PMwCAS` [51], which guarantees both the atomicity and persistence of multi-word writes. However, `PMwCAS` causes high software overhead, and it is also vulnerable to high contentions [35]. As for hash-based persistent indexes, most of them are lock-based, such as level hashing [61], CCEH [42], and CMAP [22]. P-CLHT [32] is a persistent version of CLHT [11], which supports lock-free search. Clevel hashing [6] is the concurrent version of level hashing, which uses atomic primitives to implement lock-free algorithms. However, it doesn't address the *dirty read* anomaly.

8 CONCLUSION

Existing persistent indexes suffer from low scalability and high PM overhead. Fortunately, the new platform feature for persistent memory (PM) called eADR offers opportunities to build lock-free persistent indexes and unleash the potential of PM. In this paper, we propose a lock-free PM-friendly B$^+$-Tree, named NBTree, which leverages the benefits of eADR. To achieve high scalability, NBTree develops lock-free concurrency control strategies. To reduce PM overhead, NBTree proposes a two-layer leaf node structure, which reduces PM line accesses and improves cache utilization. The real-world YCSB evaluation shows that NBTree achieves up to 11× higher throughput and 43× lower 99% tail latency than state-of-the-art persistent B$^+$-Trees.

ACKNOWLEDGMENTS

This work was supported by Natural Science Foundation of Shanghai (No. 21ZR1433600, 22ZR1435400), and Shanghai Municipal Science and Technology Major Project (No. 2021SHZDZX0102). We also thank Liangxu Nie and Yanyan Shen for their assistance and valuable feedback.

REFERENCES

[1] Andrei Alexandrescu. 2004. Generic< Programming>: Lock-Free Data Structures. In *C++ Users Journal*. Citeseer.

[2] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. 2018. BzTree: A high-performance latch-free range index for non-volatile memory. *Proceedings of the VLDB Endowment* 11, 5 (2018), 553–565.

[3] Shimin Chen and Qin Jin. 2015. Persistent b+-trees in non-volatile main memory. *Proceedings of the VLDB Endowment* 8, 7 (2015), 786–797.

[4] Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, and Jiwu Shu. 2020. uTree: a persistent B+-tree with low tail latency. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2634–2648.

[5] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. 2020. FlatStore: An efficient log-structured key-value storage engine for persistent memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 1077–1091.

[6] Zhangyu Chen, Yu Huang, Bo Ding, and Pengfei Zuo. 2020. Lock-free Concurrent Level Hashing for Persistent Memory. In *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*. 799–812.

[7] Joel Coburn, Adrian M Caulfield, Ameen Akel, Laura M Grupp, Rajesh K Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. *ACM SIGARCH Computer Architecture News* 39, 1 (2011), 105–118.

[8] Jeremy Condit, Edmund B Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 133–146.

[9] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.

[10] Tudor David, Aleksandar Dragojevic, Rachid Guerraoui, and Igor Zablotchi. 2018. Log-free concurrent data structures. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*. 373–386.

[11] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2015. Asynchronized concurrency: The secret to scaling concurrent search data structures. *ACM SIGARCH Computer Architecture News* 43, 1 (2015), 631–644.

[12] Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*. 1–15.

[13] Keir Fraser. 2004. *Practical lock-freedom*. Technical Report. University of Cambridge, Computer Laboratory.

[14] Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. 2018. A persistent lock-free queue for non-volatile memory. *ACM SIGPLAN Notices* 53, 1 (2018), 28–40.

[15] Storage Research Group. 2020. *uTree*. Tsinghua University. Retrieved Augest 26, 2021 from https://github.com/thustorage/nvm-datastructure

[16] Shashank Gugnani, Arjun Kashyap, and Xiaoyi Lu. 2020. Understanding the idiosyncrasies of real persistent memory. *Proceedings of the VLDB Endowment* 14, 4 (2020), 626–639.

[17] Jim Handy. 2015. Understanding the intel/micron 3d xpoint memory. *Proc. SDC* (2015).

[18] Timothy L Harris. 2001. A pragmatic implementation of non-blocking linked-lists. In *International Symposium on Distributed Computing*. Springer, 300–314.

[19] Timothy L Harris, Keir Fraser, and Ian A Pratt. 2002. A practical multi-word compare-and-swap operation. In *International Symposium on Distributed Computing*. Springer, 265–279.

[20] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable transient inconsistency in byte-addressable persistent b+-tree. In *16th {USENIX} Conference on File and Storage Technologies ({FAST} 18)*. 187–200.

[21] Intel. 2016. *Deprecating the PCOMMIT Instruction*. Retrieved Augest 26, 2021 from https://software.intel.com/content/www/us/en/develop/blogs/deprecate-pcommit-instruction.html

[22] Intel. 2019. *Key/Value Datastore for Persistent Memory*. Retrieved Augest 26, 2021 from https://pmem.io/pmemkv/index.html

[23] Intel. 2020. *Persistent Memory Development Kit*. Retrieved Augest 26, 2021 from http://pmem.io/pmdk

[24] Intel. 2021. *eADR: New Opportunities for Persistent Memory Applications*. Retrieved Augest 26, 2021 from https://software.intel.com/content/www/us/en/develop/articles/eadr-new-opportunities-for-persistent-memory-applications.html

[25] Intel. 2021. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Retrieved Augest 26, 2021 from https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf

[26] Tomas Karnagel, Roman Dementiev, Ravi Rajwar, Konrad Lai, Thomas Legler, Benjamin Schlegel, and Wolfgang Lehner. 2014. Improving in-memory database index performance with Intel® Transactional Synchronization Extensions. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 476–487.

[27] Wook-Hee Kim, R Madhava Krishnan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. 2021. PACTree: A High Performance Persistent Range Index Using PAC Guidelines. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM*. 424–439.

[28] R Madhava Krishnan, Wook-Hee Kim, Xinwei Fu, Sumit Kumar Monga, Hee Won Lee, Minsung Jang, Ajit Mathew, and Changwoo Min. 2021. Tips: Making volatile index structures persistent with DRAM-NVMM tiering. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*.

[29] Data Intensive Computing Lab. 2020. *FAST&FAIR*. SKKU/UNIST. Retrieved Augest 26, 2021 from https://github.com/DICL/FAST_FAIR

[30] Data-Intensive Systems Lab. 2018. *BzTree*. Simon Fraser University. Retrieved Augest 26, 2021 from https://github.com/sfu-dis/bztree

[31] Se Kwon Lee, K Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H Noh. 2017. {WORT}: Write optimal radix tree for persistent memory storage systems. In *15th {USENIX} Conference on File and Storage Technologies ({FAST} 17)*. 257–270.

[32] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. Recipe: Converting concurrent dram indexes to persistent-memory indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 462–477.

[33] Philip L Lehman and S Bing Yao. 1981. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems (TODS)* 6, 4 (1981), 650–670.

[34] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 38–49.

[35] Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. 2019. Evaluating persistent memory range indexes. *Proceedings of the VLDB Endowment* 13, 4 (2019), 574–587.

[36] Justin J Levandoski, David B Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 302–313.

[37] Jihang Liu, Shimin Chen, and Lujun Wang. 2020. LB+ Trees: Optimizing persistent index performance on 3DXPoint memory. *Proceedings of the VLDB Endowment* 13, 7 (2020), 1078–1090.

[38] Mengxing Liu. 2020. *RNTree*. Tsinghua University. Retrieved Augest 26, 2021 from https://github.com/liumx10/ICPP-RNTree

[39] Mengxing Liu, Jiankai Xing, Kang Chen, and Yongwei Wu. 2019. Building Scalable NVM-based B+ tree with HTM. In *Proceedings of the 48th International Conference on Parallel Processing*. 1–10.

[40] Shaonan Ma, Kang Chen, Shimin Chen, Mengxing Liu, Jianglang Zhu, Hongbo Kang, and Yongwei Wu. 2021. ROART: Range-query Optimized Persistent ART. In *19th {USENIX} Conference on File and Storage Technologies ({FAST} 21)*. 1–16.

[41] Maged M Michael. 2002. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*. 73–82.

[42] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H Noh, and Beomseok Nam. 2019. Write-optimized dynamic hashing for persistent memory. In *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*. 31–44.

[43] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory. In *Proceedings of the 2016 International Conference on Management of Data*. 371–386.

[44] Redis. 2009. *Redis*. Retrieved Augest 26, 2021 from https://redis.io

[45] Andy Rudoff. 2017. Persistent memory programming. *Login: The Usenix Magazine* 42, 2 (2017), 34–40.

[46] A Rudoff. 2020. Persistent memory programming without all that cache flushing. *SDC* (2020).

[47] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, Roy H Campbell, et al. 2011. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory.. In *FAST*, Vol. 11. 61–75.

[48] Vipshop. 2017. *Redis*. Retrieved Augest 26, 2021 from https://github.com/vipshop/vire

[49] Haris Volos, Andres Jaan Tack, and Michael M Swift. 2011. Mnemosyne: Lightweight persistent memory. *ACM SIGARCH Computer Architecture News* 39, 1 (2011), 91–104.

[50] Qing Wang, Youyou Lu, Junru Li, and Jiwu Shu. 2021. Nap: A black-box approach to numa-aware persistent memory indexes. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI), Virtual*.

[51] Tianzheng Wang, Justin Levandoski, and Per-Ake Larson. 2018. Easy lock-free indexing in non-volatile memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE, 461–472.

[52] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, and Jishen Zhao. 2020. Characterizing and modeling non-volatile memory systems. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 496–508.

[53] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. 2017. HiKV: A hybrid index key-value store for DRAM-NVM memory systems. In *USENIX ATC 17*. 349–362.

[54] Jian Xu and Steven Swanson. 2016. {NOVA}: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th {USENIX} Conference on File and Storage Technologies ({FAST} 16)*. 323–338.

[55] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An empirical guide to the behavior and use of scalable persistent memory. In *18th {USENIX} Conference on File and Storage Technologies ({FAST} 20)*. 169–182.

[56] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: Reducing consistency cost for NVM-based single level systems. In *13th {USENIX} Conference on File and Storage Technologies ({FAST} 15)*. 167–181.

[57] Bowen Zhang. 2021. *The consistency issue in the inner node search of FAST&FAIR*. Retrieved December 13, 2021 from https://github.com/DICL/FAST_FAIR/issues/16

[58] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. 2019. Ziggurat: a tiered file system for non-volatile main memories and disks. In *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*. 207–219.

[59] Xinjing Zhou, Lidan Shou, Ke Chen, Wei Hu, and Gang Chen. 2019. DPTree: differential indexing for persistent memory. *Proceedings of the VLDB Endowment* 13, 4 (2019), 421–434.

[60] Pengfei Zuo and Yu Hua. 2017. A write-friendly and cache-optimized hashing scheme for non-volatile memory systems. *IEEE Transactions on Parallel and Distributed Systems* 29, 5 (2017), 985–998.

[61] Pengfei Zuo, Yu Hua, and Jie Wu. 2018. Write-optimized and high-performance hashing index scheme for persistent memory. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 461–476.

[62] Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. 2019. Efficient lock-free durable sets. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–26.

TranAD: Deep Transformer Networks for Anomaly Detection in Multivariate Time Series Data

Shreshth Tuli
Imperial College London
London, UK
s.tuli20@imperial.ac.uk

Giuliano Casale
Imperial College London
London, UK
g.casale@imperial.ac.uk

Nicholas R. Jennings
Loughborough University
London, UK
n.r.jennings@lboro.ac.uk

ABSTRACT

Efficient anomaly detection and diagnosis in multivariate time-series data is of great importance for modern industrial applications. However, building a system that is able to quickly and accurately pinpoint anomalous observations is a challenging problem. This is due to the lack of anomaly labels, high data volatility and the demands of ultra-low inference times in modern applications. Despite the recent developments of deep learning approaches for anomaly detection, only a few of them can address all of these challenges. In this paper, we propose TranAD, a deep transformer network based anomaly detection and diagnosis model which uses attention-based sequence encoders to swiftly perform inference with the knowledge of the broader temporal trends in the data. TranAD uses focus score-based self-conditioning to enable robust multi-modal feature extraction and adversarial training to gain stability. Additionally, model-agnostic meta learning (MAML) allows us to train the model using limited data. Extensive empirical studies on six publicly available datasets demonstrate that TranAD can outperform state-of-the-art baseline methods in detection and diagnosis performance with data and time-efficient training. Specifically, TranAD increases F1 scores by up to 17%, reducing training times by up to 99% compared to the baselines.

PVLDB Reference Format:
Shreshth Tuli, Giuliano Casale, and Nicholas R. Jennings. TranAD: Deep Transformer Networks for Anomaly Detection in Multivariate Time Series Data. PVLDB, 15(6): 1201 - 1214, 2022.
doi:10.14778/3514061.3514067

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at `https://github.com/imperial-qore/TranAD`.

1 INTRODUCTION

Modern IT operations generate enormous amounts of high dimensional sensor data used for continuous monitoring and proper functioning of large-scale datasets. Traditionally, data mining experts have studied and highlighted data that do not follow usual trends to report faults. Such reports have been crucial for system management models for reactive fault tolerance and robust database design [47]. However, with the advent of big-data analytics and deep learning, this problem has become of interest to data mining researchers and to aid experts in handling increasing amounts of data. One particular use case is in artificial intelligence for Industry-4.0 databases, with a specific focus on service reliability [38] that has automated fault detection, recovery and management of modern systems. Detecting data-faults, or any type of behavior not conforming to the expected trends, is an active research discipline referred to as anomaly detection in multivariate time series [11]. Many data-driven industries, including ones related to distributed computing, Internet of Things (IoT), robotics and urban resource management [4, 46] are now adopting machine learning based unsupervised methods for anomaly detection.

Challenges. The problem of anomaly detection is becoming increasingly challenging in large-scale databases due to the increasing data modality [18, 28, 54]. In particular, the increasing number of sensors and devices in contemporary IoT platforms with increasing data volatility creates the requirement for significant amounts of data for accurate inference. However, due to the rising federated learning paradigm with geographically distant clusters, synchronizing databases across devices is expensive, causing limited data availability for training [48, 57]. Further, next-generation applications need ultra-fast inference speeds for quick recovery and optimal Quality of Service (QoS) [6, 50]. Time-series databases are generated using several engineering artifacts (servers, robots, *etc.*) that interact with the environment, humans or other systems. As a result, the data often displays both stochastic and temporal trends [45]. It thus becomes crucial to distinguish outliers due to stochasticity and only pinpoint observations that do not adhere to the observed temporal trends. Moreover, the lack of labeled data and anomaly diversity makes the problem challenging as we cannot use supervised learning models, which have shown to be effective in other areas of data mining [12]. Finally, it is not only important to detect anomalies but also the root causes, *i.e.*, the specific data sources leading to abnormal behavior [23]. This complicates the problem further as we need to perform multi-class prediction (whether there is an anomaly and from which source if so) [60].

Existing solutions. The above discussed challenges have led to the development of a myriad of unsupervised learning solutions for automated anomaly detection. Researchers have developed reconstruction-based methods that predominantly aim to encapsulate the temporal trends and predict the time-series data in an unsupervised fashion, then use the deviation of the prediction with the ground-truth data as anomaly scores. Based on various extreme value analysis methods, such approaches classify timestamps with high anomaly scores as abnormal [4, 10, 14, 20, 28, 29, 45, 60, 62]. The way prior works generate a predicted time-series from a given one varies from one work to another. Traditional approaches, like SAND [10], use clustering and statistical analysis to

Proceedings of the VLDB Endowment, Vol. 15, No. 6 ISSN 2150-8097.
doi:10.14778/3514061.3514067

detect anomalies. Contemporary methods like openGauss [30] and LSTM-NDT [20] use a Long-Short-Term-Memory (LSTM) based neural networks to forecast the data with an input time-series and a non-parametric dynamic thresholding approach for detecting anomalies from prediction errors. However, recurrent models like LSTMs are known to be slow and computationally expensive [4]. Recent state-of-the-art methods, like MTAD-GAT [62] and GDN [14], use deep neural networks with a time-series window as an input for more accurate predictions. However, as the inputs become more data-intensive, small constant size window inputs limit the detection performance of such models due to the restricted local context information given to the model [4]. There is a need for a model that is fast and can capture high-level trends with minimal overheads.

New insights. As noted above, recurrent models based on prior methods are not only slow and computationally expensive, but are also unable to model long-term trends effectively [4, 14, 62]. This is because, at each timestamp, a recurrent model needs to first perform inference for all previous timestamps before proceeding further. Recent developments of the transformer models allow single-shot inference with the complete input series using position encoding [51]. Using transformers allows much faster detection compared to recurrent methods by parallelizing inference on GPUs [19]. However, transformers also provide the benefit of being able to encode large sequences with accuracy and training/inference times nearly agnostic to the sequence length [51]. Thus, we use transformers to grow the temporal context information sent to an anomaly detector without significantly increasing the computational overheads.

Our contributions. This work uses various tools, including Transformer neural networks and model-agnostic meta learning, as building blocks. However, each of these different technologies cannot be directly used and need necessary adaptations to create a generalizable model for anomaly detection. Specifically, we propose a transformer-based anomaly detection model (TranAD), that uses self-conditioning and an adversarial training process. Its architecture makes it fast for training and testing while maintaining stability with large input sequences. Simple transformer-based encoder-decoder networks tend to miss anomalies if the deviation is too small, *i.e.*, it is relatively close to normal data. One of our contributions is to show that this can be alleviated by an adversarial training procedure that can amplify reconstruction errors. Further, using self-conditioning for robust multi-modal feature extraction can help gain training stability and allow generalization [32]. This, with model-agnostic meta learning (MAML) helps keep optimum detection performance even with limited data [15], as we show later in the validation that methods with simple transformers underperform by over 11% compared to TranAD. We perform extensive empirical experiments on publicly available datasets to compare and analyze TranAD against the state-of-the-art methods. Our experiments show that TranAD is able to outperform baselines by increasing prediction scores by up to 17% while reducing training time overheads by up to 99%.

The rest of the paper is organized as follows. Section 2 overviews related work. Section 3 outlines the working of the TranAD model for multivariate anomaly detection and diagnosis. A performance evaluation of the proposed method is shown in Section 4. Section 5 presents additional analysis. Finally, Section 6 concludes.

2 RELATED WORK

Time series anomaly detection is a long-studied problem in the VLDB community. The prior literature works on two types of time-series data: univariate and multivariate. For the former, various methods analyze and detect anomalies in time-series data with a single data source [34], while for the latter multiple time-series together [14, 45, 62].

Classical methods. Such methods for anomaly detection typically model the time-series distribution using various classical techniques like k-Mean clustering, Support Vector Machines (SVMs) or regression models [10, 28, 43, 52]. Other methods use wavelet theory or various signal transformation methods like Hilbert transform [25]. Other classes of methods use Principal Component Analysis (PCA), process regression or hidden Markov chains to model time-series data [41]. The GraphAn technique [9] converts the time-series inputs to graphs and uses graph distance metrics to detect outliers. Another technique, namely isolation forest, uses an ensemble of several isolation trees that recursively partition the feature space for outlier detection [5, 31]. Finally, classical methods use variants of Auto-Regressive Integrated Moving Average (ARIMA) to model and detect anomalous behaviour [56]. However, auto-regression based approaches are rarely used for anomaly detection in high-order multivariate time series due to their inability to efficiently capture volatile time-series [1]. Other methods like SAND [10], CPOD [47] and Elle [28] utilize clustering and database read-write history to detect outliers.

Time-series discord discovery is another recently proposed method for fault prediction [16, 37, 58, 59]. Time series discords refer to the most unusual time series subsequences, *i.e.*, subsequences that are maximally different from all other subsequences in the same time series. A sub-class of methods uses matrix profiling or its variants for anomaly and motif discovery by detecting time series discords [16, 35, 63]. Many advances have been proposed to make matrix profiling techniques data and time efficient [21]. Other efforts aim to make matrix profiling applicable to diverse domains [64]. However, matrix profiling has many more uses than just anomaly detection and is considered to be slower than pure discord discovery algorithms [37]. A recent approach, MERLIN [37], uses a parameter-free version of time series discord discovery by iteratively comparing subsequences of varying length with their immediate neighbors. MERLIN is considered to be the state-of-the-art discord discovery approach with low overheads; hence, is regarded as one of the baselines in our experiments.

Deep Learning based methods. Most contemporary state-of-the-art techniques employ some form of deep neural networks. The LSTM-NDT [20] method relies on an LSTM based deep neural network model that uses the input sequence as training data and, for each input timestamp, forecasts data for the next timestamp. LSTMs are auto-regressive neural networks that learn order dependence in sequential data, where the prediction at each timestamp uses feedback from the output of the previous timestamp. This work also proposes a non-parametric dynamic error thresholding (NDT) strategy to set a threshold for anomaly labeling using moving averages of the error sequence. However, being a recurrent model, such models are slow to train in many cases with long input sequences.

Further, LSTMs are often inefficient in modeling long temporal patterns, especially when the data is noisy [62].

The DAGMM [65] method uses a deep autoencoding Gaussian mixture model for dimension reduction in the feature space and recurrent networks for temporal modeling. This work predicts an output using a mixture of Gaussians, where the parameters of each Gaussian are given by a deep neural model. The autoencoder compresses an input datapoint into a latent space, that is then used by a recurrent estimation network to predict the next datapoint. The decoupled training of both networks allows the model to be more robust; however, it still is slow and unable to explicitly utilize inter-modal correlations [14]. The Omnianomaly [45] uses a stochastic recurrent neural network (similar to an LSTM-Variational Autoencoder [39]) and a planar normalizing flow to generate reconstruction probabilities. It also proposes an adjusted Peak Over Threshold (POT) method for automated anomaly threshold selection that outperforms the previously used NDT approach. This work led to a significant performance leap compared to the prior art, but at the expense of high training times.

The Multi-Scale Convectional Recursive Encoder-Decoder (MSCRED) [60] converts an input sequence window into a normalized two-dimensional image and then passes it through a ConvLSTM layer. This method is able to capture more complex inter-modal correlations and temporal information, however is unable to generalize to settings with insufficient training data. The MAD-GAN [29] uses an LSTM based GAN model to model the time-series distribution using generators. This work uses not only the prediction error, but also the discriminator loss in the anomaly scores. MTAD-GAT [62] uses a graph-attention network to model both feature and temporal correlations and pass it through a lightweight Gated-Recurrent-Unit (GRU) network that aids detection without severe overheads. Traditionally, attention operations perform input compression using convex combination where the weights are determined using neural networks. GRU is a simplified version of LSTM with a smaller parameter set and can be trained in limited data settings. The CAE-M [61] uses a convolutional autoencoding memory network, similar to MSCRED. It passes the time-series through a CNN with the output being processed by bidirectional LSTMs to capture long-term temporal trends. Such recurrent neural network-based models have been shown to have high computation costs and low scalability for high dimensional datasets [4].

More recent works such as USAD [4], GDN [14] and openGauss [30] do not use resource-hungry recurrent models, but only attention-based network architectures to improve training speeds. The USAD method uses an autoencoder with two decoders with an adversarial game-style training framework. This is one of the first works that focus on low overheads by using a simple autoencoder and can achieve a several-fold reduction in training times compared to the prior art. The Graph Deviation Network (GDN) approach learns a graph of relationships between data modes and uses attention-based forecasting and deviation scoring to output anomaly scores. The openGauss approach uses a tree-based LSTM that has lower memory and computational footprint and allows capturing temporal trends even with noisy data. However, due to the small window as an input and the use of simple or no recurrent models, the latest models are unable to capture long-term dependencies effectively.

The recently proposed HitAnomaly [19] method uses vanilla transformers as encoder-decoder networks, but is only applicable to natural-language log data and not appropriate for generic continuous time-series data as inputs. In our experiments, we compare TranAD against the state-of-the-art methods MERLIN, LSTM-NDT, DAGMM, OmniAnomaly, MSCRED, MAD-GAN, USAD, MTAD-GAT, CAE-M and GDN. These methods have shown superiority in anomaly detection and diagnosis, but complement one another in terms of performance across different time-series datasets. Out of these, only USAD aims to reduce training times, but does this to a limited extent. Just like reconstruction based prior work [4, 29, 45, 60, 61], we develop a TranAD model that learns broad level trends using training data to find anomalies in test data. We specifically improve anomaly detection and diagnosis performance with also reducing the training times in this work.

3 METHODOLOGY

3.1 Problem Formulation

We consider a multivariate time-series, which is a timestamped sequence of observations/datapoints of size T

$$\mathcal{T} = \{x_1, \ldots, x_T\},$$

where each datapoint x_t is collected at a specific timestamp t and $x_t \in \mathbb{R}^m$, $\forall t$. Here, the univariate setting is a particular case where $m = 1$. We now define the two problems of anomaly detection and diagnosis.

Anomaly Detection: Given a training input time-series $\mathcal{T}$, for any unseen test time-series $\hat{\mathcal{T}}$ of length $\hat{T}$ and same modality as the training series, we need to predict $\mathcal{Y} = \{y_1, \ldots, y_{\hat{T}}\}$, where we use $y_t \in \{0, 1\}$ to denote whether the datapoint at the t-th timestamp of the test set is anomalous (1 denotes an anomalous datapoint).

Anomaly Diagnosis: Given the above training and test time-series, we need to predict $\mathcal{Y} = \{y_1, \ldots, y_{\hat{T}}\}$, where $y_t \in \{0, 1\}^m$ to denote which of the modes of the datapoint at the t-th timestamp are anomalous.

3.2 Data Preprocessing

To make our model more robust, we normalize the data and convert it to time-series windows, both for training and testing. We normalize the time-series as:

$$x_t \leftarrow \frac{x_t - \min(\mathcal{T})}{\max(\mathcal{T}) - \min(\mathcal{T}) + \epsilon'}, \tag{1}$$

where $\min(\mathcal{T})$ and $\max(\mathcal{T})$ are the mode wise minimum and maximum vectors in the training time-series. ϵ' is a small constant vector to prevent zero-division. Knowing the ranges a-priori, we normalize the data to get it in the range $[0, 1)$.

To model the dependence of a data point x_t at a timestamp t, we consider a local contextual window of length K as

$$W_t = \{x_{t-K+1}, \ldots, x_t\}.$$

We use replication padding for $t < K$ and convert an input time series $\mathcal{T}$ to a sequence of sliding windows $\mathcal{W} = \{W_1, \ldots, W_T\}$. Replication padding, for each $t < K$, appends to the window W_t a constant vector $\{x_t, \ldots, x_t\}$ of length $K - t$ to maintain the window length of K for each t. Instead of using $\mathcal{T}$ as training input, we use $\mathcal{W}$ for model training and $\hat{\mathcal{W}}$ (corresponding to $\hat{\mathcal{T}}$) as the test

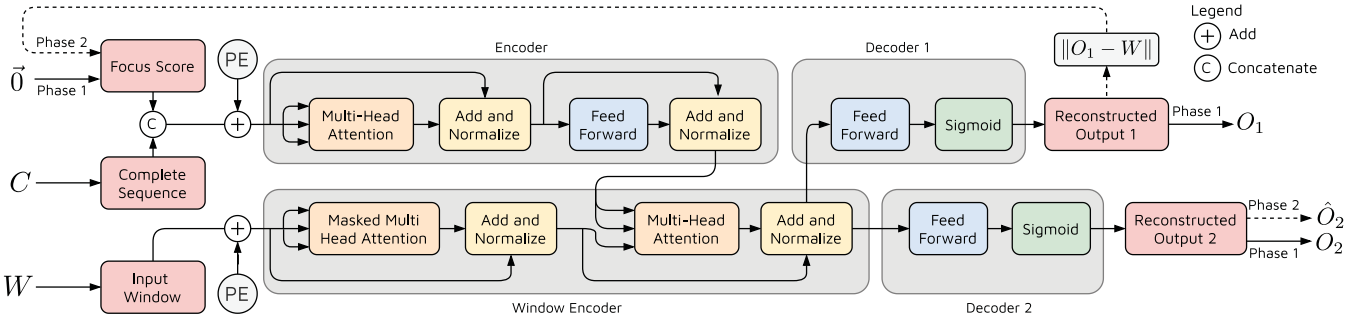


Figure 1: The TranAD Model.

series. This is a common practice in prior work [4, 45] as it allows us to give a datapoint with its local context instead of a standalone vector, and hence is used in our model. We also consider the time slice until the current timestamp t of a series $\mathcal{T}$ and denote it as C_t.

Now, instead of directly predicting the anomaly label y_t for each input window W_t, we shall first predict an anomaly score s_t for this window. Using anomaly scores for the past input windows, we calculate a threshold value D, above which we label the input window as anomalous, thus $y_t = \mathbb{1}(s_t \geq D)$. To calculate the anomaly score s_t, we reconstruct the input window as O_t and use the deviation between W_t and O_t. For the sake of simplicity and without loss of generality, we shall use W, C, O and s for the rest of the discussion.

3.3 Transformer Model

Transformers are popular deep learning models that have been used in various natural language and vision processing tasks [51]. However, we use insightful refactoring of the transformer architecture for the task of anomaly detection in time-series data. Just like other encoder-decoder models, in a transformer, an input sequence undergoes several attention-based transformations. Figure 1 shows the architecture of the neural network used in TranAD. The encoder encodes the complete sequence until the current timestamp C with a focus score (more details later). The window encoder uses this to create an encoded representation of the input window W, which is then passed to two decoders to create its reconstruction.

We now provide details on the working of TranAD. A multivariate sequence like W or C is transformed first into a matrix form with modality m. We define scaled-dot product attention [51] of three matrices Q (query), K (key) and V (value):

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{m}}\right)V. \tag{2}$$

Here, the softmax forms the convex combination weights for the values in V, allowing us to compress the matrix V into a smaller representative embedding for simplified inference in the downstream neural network operations. Unlike traditional attention operation, the scaled-dot product attention scales the weights by a $\sqrt{m}$ term to reduce the variance of the weights, facilitating stable training [51]. For input matrices Q, K and V, we apply Multi-Head Self Attention [51] by first passing it through h (number of heads) feed-forward layers to get Q_i, K_i and V_i for $i \in \{1, \ldots, h\}$, and then applying scaled-dot product attention as

$$\begin{aligned} \text{MultiHeadAtt}(Q, K, V) &= \text{Concat}(H_1, \ldots, H_h), \\ \text{where } H_i &= \text{Attention}(Q_i, K_i, V_i). \end{aligned} \tag{3}$$

Multi-Head Attention allows the model to jointly attend to information from different representation sub-spaces at different positions. In addition, we use position encoding of the input matrices as defined in [51].

As GAN models have been shown to perform well in characteristic tasks of whether an input is anomalous or not, we leverage a time-efficient GAN style adversarial training method. Our model consists of two transformer encoders and two decoders (Figure 1). We consider the model inference in two phases. We first take the W and C pair as an input and a focus score F (initially a zero matrix of the dimension of W, more details in the next subsection). We broadcast F to match the dimension of W, with appropriate zero-padding and concatenate the two. We then apply position encoding and obtain the input for the first encoder, say I_1. The first encoder performs the following operations

$$\begin{aligned} I_1^1 &= \text{LayerNorm}(I_1 + \text{MultiHeadAtt}(I_1, I_1, I_1)), \\ I_1^2 &= \text{LayerNorm}(I_1^1 + \text{FeedForward}(I_1^1)). \end{aligned} \tag{4}$$

Here, $\text{MultiHeadAtt}(I_1, I_1, I_1)$ denotes the multi-head self attention operation for the input matrix I_1 and $+$ denotes matrix addition. The above operations generate attention weights using the input time-series windows and the complete sequence to capture temporal trends within the input sequences. These operations enable the model to infer over multiple batches of the time-series windows in parallel as the neural network, at each timestamp, does not depend on the output of a previous timestamp, significantly improving the training time of the proposed method. For the window encoder, we apply position encoding to the input window W to get I_2. We modify the self-attention in the window encoder to mask the data at subsequent positions. This is done to prevent the decoder from looking at the datapoints for future timestamp values at the time of training as all data W and C is given at once to allow parallel training. The window encoder performs the following operations

$$\begin{aligned} I_2^1 &= \text{Mask}(\text{MultiHeadAtt}(I_2, I_2, I_2)), \\ I_2^2 &= \text{LayerNorm}(I_2 + I_2^1), \\ I_2^3 &= \text{LayerNorm}(I_2^2 + \text{MultiHeadAtt}(I_1^2, I_1^2, I_2^2)). \end{aligned} \tag{5}$$

Algorithm 1 The TranAD training algorithm

Require:
Encoder E, Decoders D_1 and D_2
Dataset used for training $\mathcal{W}$
Evolutionary hyperparameter ϵ
Iteration limit N

Initialize weights E, D_1, D_2
$n \leftarrow 0$
do
  for$(t = 1$ to $T)$
    $O_1, O_2 \leftarrow D_1(E(W_t, \vec{0})), D_2(E(W_t, \vec{0}))$
    $\hat{O}_2 \leftarrow D_2(E(W_t, \|O_1 - W_t\|_2))$
    $L_1 = \epsilon^{-n}\|O_1 - W_t\|_2 + (1 - \epsilon^{-n})\|\hat{O}_2 - W_t\|_2$
    $L_2 = \epsilon^{-n}\|O_2 - W_t\|_2 - (1 - \epsilon^{-n})\|\hat{O}_2 - W_t\|_2$
    Update weights of E, D_1, D_2 using L_1, L_2
    $n \leftarrow n + 1$
  Meta-Learn weights E, D_1, D_2 using a random batch
while $n < N$

The encoding of the complete sequence I_1^2 is used as value and keys by the window encoder for the attention operation using the encoded input window as the query matrix. The motivation behind the operations in (5) is similar to the one for (4); however, here we apply masking of the window input to hide the window sequences for future timestamps in the same input batch. As the complete input sequence up to the t-th timestamp is given to the model as an input; it allows the model to encapsulate and leverage a larger context compared to a bounded, limited one as in prior art [4, 45, 62]. Finally, we use two identical decoders which perform the operation

$$O_i = \text{Sigmoid}(\text{FeedForward}(I_2^3)), \tag{6}$$

where $i \in \{1, 2\}$ for the first and second decoder respectively. The Sigmoid activation is used to generate an output in the range $[0, 1]$, to match the normalized input window. Thus, the TranAD model takes the inputs C and W to generate two outputs O_1 and O_2.

3.4 Offline Two-Phase Adversarial Training

We now describe the adversarial training process and the two-phase inference approach in the TranAD model, summarized in Algorithm 1.

Phase 1 - Input Reconstruction. The Transformer model enables us to predict the reconstruction of each input time-series window. It does this by acting as an encoder-decoder network at each timestamp. However, traditional encoder-decoder models often are unable to capture short-term trends and tend to miss anomalies if the deviations are too small [29]. To tackle this challenge, we develop an auto-regressive inference style that predicts the reconstructed window in two-phases. In the first phase, the model aims to generate an approximate reconstruction of the input window. The deviation from this inference, referred to as the *focus score* mentioned previously, facilitates the attention network inside the Transformer Encoder to extract temporal trends, focusing on the sub-sequences where the deviations are high. Thus, the output of the second phase is conditioned on the deviations generated from the first phase. Thus, in the first stage, the encoders convert the input window $W \in \mathbb{R}^{K \times m}$ (with focus score $F = [0]_{K \times m}$) to a compressed latent representation I_2^3 using context-based attention as in a common transformer model. This compressed representation is then converted to generate outputs O_1 and O_2 via Eq. (6).

Phase 2 - Focused Input Reconstruction. In the second phase, we use the reconstruction loss for the first decoder as a focus score. Having the focus matrix for the second phase $F = L_1$, we rerun model inference to obtain the output of the second decoder as $\hat{O}_2$.

The focus score generated in the first phase indicates the deviations of the reconstructed output from the given input. This acts as a prior to modify the attention weights in the second phase and gives higher neural network activation to specific input sub-sequences to extract short-term temporal trends. We refer to this approach as "self-conditioning" in the rest of the paper. This two-phase auto-regressive inference style has a three-fold benefit. First, it amplifies the deviations, as the reconstruction error acts as an activation in the attention part of the Encoder in Figure 1, to generate an anomaly score, simplifying the fault-labeling task (discussed in Section 3.5). Second, it prevents false positives by capturing short-term temporal trends in the Window Encoder in Figure 1. Third, the adversarial style training is known to improve generalizability and make the model robust to diverse input sequences [4].

Evolving Training Objective. The above-described model is bound to suffer from similar challenges as in other adversarial training frameworks. One of the critical challenges is maintaining training stability. To tackle this, we design an adversarial training procedure that uses outputs from two separate decoders (Decoders 1 and 2 in Figure 1). Initially, both decoders aim to independently reconstruct the input time-series window. As in [45] and [39], we define the reconstruction loss for each decoder using the L2-norm using the outputs of the first phase:

$$\begin{aligned} L_1 &= \|O_1 - W\|_2, \\ L_2 &= \|O_2 - W\|_2. \end{aligned} \tag{7}$$

We now introduce the adversarial loss that uses outputs of the second phase. Here, the second decoder aims to distinguish between the input window and the candidate reconstruction generated by the first decoder in phase 1 (using the focus scores) by maximizing the difference $||\hat{O}_2 - W||_2$. On the other hand, the first decoder aims to fool the second decoder by aiming to create a degenerate focus score (a zero vector) by perfectly reconstructing the input (*i.e.*, $O_1 = W$). This pushes the decoder 2, in this phase, to generate the same output as O_2 which it aims to match the input in phase 1. This means the training objective is

$$\min_{\text{Decoder1}} \max_{\text{Decoder2}} \|\hat{O}_2 - W\|_2. \tag{8}$$

Thus, the objective of the first decoder is to minimize the reconstruction error of this self-conditioned output, whereas the objective of the second one is to maximize the same. We realize this by using the loss as:

$$\begin{aligned} L_1 &= +\|\hat{O}_2 - W\|_2, \\ L_2 &= -\|\hat{O}_2 - W\|_2. \end{aligned} \tag{9}$$

Now that we have loss functions for both phases, we need to determine the cumulative loss for each decoder. We thus use an evolutionary loss function that combines the reconstruction and

Algorithm 2 The TranAD testing algorithm

Require:
Trained Encoder E, Decoders D_1 and D_2
Test Dataset $\hat{\mathcal{W}}$

1: **for**($t = 1$ to $\hat{T}$)
2: $O_1, O_2 \leftarrow D_1(E(\hat{W}_t, \vec{0})), D_2(E(\hat{W}_t, \vec{0}))$
3: $\hat{O}_2 \leftarrow D_1(E(\hat{W}_t, \|O_1 - W\|_2)), D_2(E(\hat{W}_t, \|O_1 - W\|_2))$
4: $s = \frac{1}{2}\|O_1 - \hat{W}\|_2 + \frac{1}{2}\|\hat{O}_2 - \hat{W}\|_2$
5: $y_i = \mathbb{1}(s_i \geq \text{POT}(s_i))$
6: $y = \bigvee_i y_i$

adversarial loss functions from the two phases as

$$\begin{aligned} L_1 &= \epsilon^{-n}\|O_1 - W\|_2 + (1 - \epsilon^{-n})\|\hat{O}_2 - W\|_2, \\ L_2 &= \epsilon^{-n}\|O_2 - W\|_2 - (1 - \epsilon^{-n})\|\hat{O}_2 - W\|_2, \end{aligned} \tag{10}$$

where n is the training epoch and ϵ is a training parameter close to one (lines 7-8 in Alg. 1). Initially, the weight given to the reconstruction loss is high. This is to ensure stable training when the outputs of the decoders are poor reconstructions of the input window. With poor reconstructions, the focus scores used in the second phase would be unreliable; and hence, cannot be utilized as a prior to indicating reconstructions that are far from the input sequence. Thus, the adversarial loss is given a low weight in the initial part of the process to avoid destabilizing model training. As reconstructions become closer to the input windows, and focus scores become more precise, the weight to the adversarial loss is increased. As loss curves in the neural network training process typically follow exponential function, we use weights of the form ϵ^{-n} in the training process with a small positive constant ϵ.

As the training process does not assume that the data is available sequentially (as in an online process), the complete time-series can be split into (W, C) pairs and the model can be trained using input batches. Masked multi-head attention allows us to run this in parallel across several batches and speed up the training process.

Meta Learning. Finally, our training loop uses model-agnostic meta learning (MAML), a few-shot learning model for fast adaptation of neural networks [15]. This helps our TranAD model learn temporal trends in the input training time-series with limited data. In each training epoch, a gradient update for neural network weights (without loss in generality assume θ) can be simply written

$$\theta' \leftarrow \theta - \alpha\nabla_\theta L(f(\theta)), \tag{11}$$

where α, $f(\cdot)$ and $L(\cdot)$ are learning rate, abstract representation of the neural network and loss function respectively. Now, at the end of each epoch we perform meta-learning step as

$$\theta \leftarrow \theta - \beta\nabla_\theta L(f(\theta')). \tag{12}$$

The meta-optimization is performed with a meta step-size β, over the model weights θ where the objective is evaluated using the updated weights θ'. Prior work has shown that this allows models to be trained quickly with limited data [15]. We encapsulate this in a single line in Algorithm 1 (line 11).

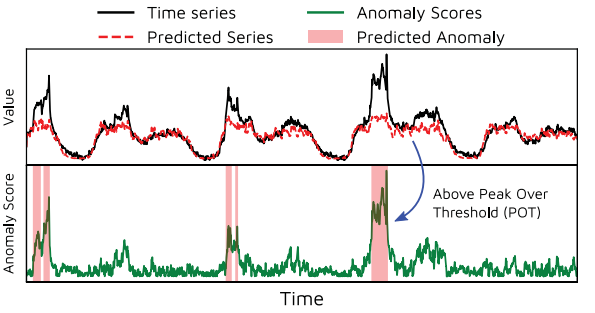


Figure 2: Visualization of anomaly prediction.

3.5 Online Inference, Anomaly Detection and Diagnosis

We now describe the inference procedure using the trained transformer model (summarized in Algorithm 2). For an unseen data ($\hat{W}$, $\hat{C}$), the anomaly score is defined as

$$s = \frac{1}{2}\|O_1 - \hat{W}\|_2 + \frac{1}{2}\|\hat{O}_2 - \hat{W}\|_2. \tag{13}$$

The inference at test time runs again in two phases and hence we get a single pair of reconstruction $(O_1, \hat{O}_2)$ (lines 2 and 3 in Alg. 2). At test time, we only consider the data until the current timestamp and hence this operation runs sequentially in an online fashion. Once we have the anomaly scores for a timestamp for each dimension s_i, we label the timestamp anomalous if this score is greater than a threshold. As is common in prior work [9, 20, 45], for fair comparison, we use the Peak Over Threshold (POT) [44] method to choose the threshold automatically and dynamically. In essence, this is a statistical method that uses "extreme value theory" to fit the data distribution with a Generalized Pareto Distribution and identify appropriate value at risk to dynamically determine threshold values. We also tested with another popular EVT method, namely annual maximum (AM) [7]; however, we have observed 7.2% higher F1 scores on an average for TranAD with POT than AM. Anomaly diagnosis label for each dimension i (y_i) and detection (y) results is defined as

$$\begin{aligned} y_i &= \mathbb{1}(s_i \geq \text{POT}(s_i)), \\ y &= \bigvee_i y_i. \end{aligned} \tag{14}$$

Thus, we label the current timestamp anomalous if any of the m dimensions is anomalous (lines 5-6 in Alg. 2). Figure 2 illustrates this process for a sample time-series.

Impact of Attention and Focus Scores. Figure 3 visualizes the attention and focus scores for the TranAD model trained on the SMD dataset (details in Section 4.1). We show the time-series, the average attention weights for each window (averaged over multiple heads) and focus scores for the first six dimensions of the dataset. It is apparent that the focus scores are highly correlated with the peaks and noise in the data. There is also a high correlation of focus scores across dimensions. For timestamps with sudden changes in the time-series, focus scores are higher. Further, the model gives higher attention weights to the specific dimensions of the time-series where the deviations are higher. This allows the model to specifically detect anomalies in each dimension individually, with the contextual trend of the complete sequence as a prior.

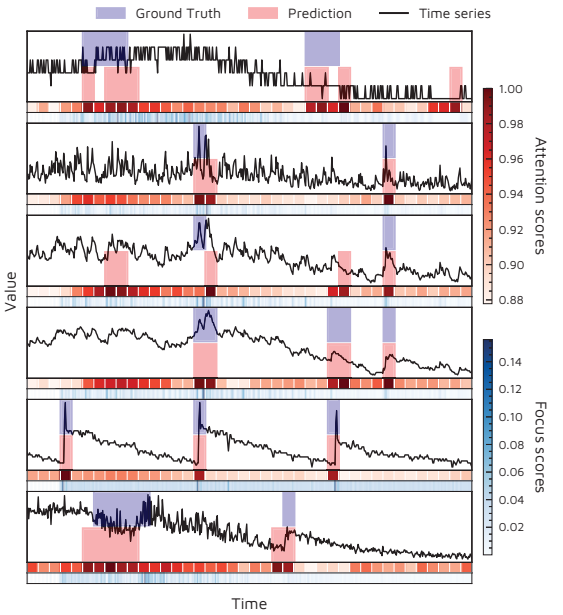


Figure 3: Visualization of focus and attention scores.

4 EXPERIMENTS

We compare TranAD with state-of-the-art models for mutlivariate time-series anomaly detection, including MERLIN [37], LSTM-NDT [20] (with autoencoder implementation from openGauss [30]), DAGMM [65], OmniAnomaly [45], MSCRED [60], MAD-GAN [29], USAD [4], MTAD-GAT [62], CAE-M [61] and GDN [14] (with graph embedding implementation from GraphAn [9]) . For more details refer Section 2.[1] We also tested the Isolation Forest method, but due to its low F1 scores, do not include the corresponding results in our discussion. Other classical methods have been omitted as deep-learning based approaches have already been shown to outperform them in prior work [4, 14, 62]. We use hyperparameters of the baseline models as presented in their respective papers. We train all models using PyTorch-1.7.1 [40] library[2].

We use the AdamW [27] optimizer to train our model with an initial learning rate of 0.01 (meta learning rate 0.02) and step-scheduler with step size of 0.5 [42]. We use the following hyperparameter values determined using grid-search.

- Window size = 10.
- Number of layers in transformer encoders = 1
- Number of layers in feed-forward unit of encoders = 2
- Hidden units in encoder layers = 64
- Dropout in encoders = 0.1

[1]We use publicly available code sources for most of the baselines. LSTM-NDT `https://github.com/khundman/telemanom`, openGauss `https://gitee.com/opengauss/openGauss-AI`, DAGMM `https://github.com/tnakae/DAGMM`, OmniAnomaly `https://github.com/NetManAIOps/OmniAnomaly`, MSCRED `https://github.com/7fantasysz/MSCRED`, MAD-GAN `https://github.com/LiDan456/MAD-GANs`. All URLs last accessd on 18 February 2022. Other models were re-implemented by us (details on the implementation of the MERLIN baseline in [49]).
[2]Parallel Transformer training was implemented as per [51]. All model training and experiments were performed on a system with configuration: Intel i7-10700K CPU, 64GB RAM, Nvidia RTX 3080 and Windows 11 OS.

Table 1: Dataset Statistics

Dataset	Train	Test	Dimensions	Anomalies (%)
NAB	4033	4033	1 (6)	0.92
UCR	1600	5900	1 (4)	1.88
MBA	100000	100000	2 (8)	0.14
SMAP	135183	427617	25 (55)	13.13
MSL	58317	73729	55 (3)	10.72
SWaT	496800	449919	51 (1)	11.98
WADI	1048571	172801	123 (1)	5.99
SMD	708405	708420	38 (4)	4.16
MSDS	146430	146430	10 (1)	5.37

The effect of window size on anomaly detection performance is analyzed in Section 5. We choose hyperparameters other than the window size using grid search. For POT parameters, coefficient = 10^{-4} for all data sets, low quantile is 0.07 for SMAP, 0.01 for MSL, and 0.001 for others. These were selected as per the implementation of the OmniAnomaly baseline [45]. The only dataset-specific hyperparameter is the number of heads in multi-head attention, which was kept to be the same as the dimension size of the dataset. Other assignments for this hyperparameter give similar broad-level trends.

To train TranAD, we divide the training time-series into 80% training data and 20% validation data. To avoid model over-fitting, we use early-stopping criteria to train TranAD, *i.e.*, we stop the training process once the validation accuracy starts to decrease.

4.1 Datasets

We use seven publicly available datasets in our experiments. We summarize their characteristics in Table 1. The values in parenthesis are the number of sequences in the dataset repository and we report average scores across all sequences in a dataset. For instance, the SMAP dataset has 55 traces with 25 dimensions each. While we share some of the concerns expressed in [55] about the lack of quality benchmark datasets for time series anomaly detection, we use these commonly-used benchmark datasets here to enable direct comparison of our approach to competing methods.

(1) *Numenta Anomaly Benchmark (NAB)*: is a dataset of multiple real-world data traces, including readings from temperature sensors, CPU utilization of cloud machines, service request latencies and taxi demands in New York city [2]. However, this dataset is known to have sequences with incorrect anomaly labels [55] such as the `nyc-taxi` trace [37], which we exclude in our experiments.
(2) *HexagonML (UCR) dataset*: is a dataset of multiple univariate time series (included just for completeness) that was used in KDD 2021 cup [13, 26]. We include only the datasets obtained from natural sources (the `InternalBleeding` and `ECG` datasets) and ignore the synthetic sequences.
(3) *MIT-BIH Supraventricular Arrhythmia Database (MBA)*: is a collection of electrocardiogram recordings from four patients, containing multiple instances of two different kinds of anomalies (either supraventricular contractions or premature heartbeats) [17, 36]. This is a popular large-scale dataset in the data management community [8, 10].

(4) *Soil Moisture Active Passive (SMAP) dataset*: is a dataset of soil samples and telemetry information using the Mars rover by NASA [20].
(5) *Mars Science Laboratory (MSL) dataset*: is a dataset similar to SMAP but corresponds to the sensor and actuator data for the Mars rover itself [20]. However, this dataset is known to have many trivial sequences [55]; hence, we consider only the three non-trivial ones (`A4`, `C2` and `T1`) pointed out by [37].
(6) *Secure Water Treatment (SWaT) dataset*: This dataset is collected from a real-world water treatment plant with 7 days of normal and 4 days of abnormal operation [33]. This dataset consists of sensor values (water level, flow rate, etc.) and actuator operations (valves and pumps).
(7) *Water Distribution (WADI) dataset*: This is an extension of the SWaT system but had more than twice the number of sensors and actuators than the SWaT model [3]. The dataset is also collected for a longer duration of 14 and 2 days of normal and attack scenarios.
(8) *Server Machine Dataset (SMD)*: This is a five-week long dataset of stacked traces of the resource utilizations of 28 machines from a compute cluster [45]. Similar to MSL, we use the non-trivial sequences in this dataset, specifically the traces named `machine-1-1`, `2-1`, `3-2` and `3-7`.
(9) *Multi-Source Distributed System (MSDS) Dataset*: This is a recent high-quality multi-source data composed of distributed traces, application logs, and metrics from a complex distributed system [38]. This dataset is specifically built for AI operations, including automated anomaly detection, root cause analysis, and remediation.

We eschew comparisons on the Yahoo [53] dataset that has been claimed to suffer from mislabeling and run-to-failure bias [55].

4.2 Evaluation Metrics

4.2.1 Anomaly Detection. We use precision, recall, area under the receiver operating characteristic curve (ROC/AUC) and F1 score to evaluate the detection performance of all models [14, 62]. We also measure the AUC and F1 scores by training all models with 20% of the training data (again using the 80:20 split for validation dataset and the rest as the test set), and call these AUC* and F1* respectively, to measure the performance of the models with limited data. We train on the five sets of 20% training data and report average results for statistical significance.

4.2.2 Anomaly Diagnosis. We use commonly used metrics to measure the diagnosis performance of all models [62]. HitRate@P% is the measure of how many ground truth dimensions have been included in the top candidates predicted by the model [45]. *P%* is the percentage of the ground truth dimensions for each timestamp, which we use to consider the top predicted candidates. For instance, if at timestamp t, if 2 dimensions are labeled anomalous in the ground truth, HitRate@100% would consider top 2 dimensions and HitRate@150% would consider 3 dimensions (100 and 150 are chosen based on prior work [62]). We also measure the Normalized Discounted Cumulative Gain (NDCG) [24]. NDCG@P% considers the same number of top predicted candidates as HitRate@P%.

4.3 Results

Anomaly Detection. Tables 2 and 3 provide the precision, recall, AUC, F1, AUC* and F1* scores for TranAD and baseline models for all datasets. On average, the F1 score of the TranAD model is 0.8802 and F1* is 0.8012. TranAD outperforms the baselines (in terms of F1 score) for all datasets except MSL when we consider the complete dataset for model training. TranAD also outperforms baselines for all datasets except the WADI dataset with 20% of the dataset used for training (F1* score). For MSL, the GDN model has the highest F1 score (0.9591) and for the WADI dataset, OmniAnomaly has the highest F1* score (0.1017). Similarly, TranAD outperforms baselines in terms of AUC scores for all datasets except MSDS, where GDN has the highest AUC (0.9105). All models perform relatively poorly on WADI due to its large-scale in terms of sequence lengths and data modality. Specifically, TranAD achieves improvement of up to 17.06% in F1 score, 14.64% in F1* score, 11.69% in AUC and 11.06% in AUC* scores over the state-of-the-art baseline models.

The MERLIN baseline is a parameter free approach that does not require any training data; hence, we report F1* and AUC* as F1 and AUC scores, respectively. MERLIN performs relatively well only on the univariate datasets, *i.e.* NAB and UCR, and is unable to scale effectively to multivariate data in our traces. The baseline method LSTM-NDT has a good performance on MSL and SMD, but performs poorly on other datasets. This is due to its sensitivity to different scenarios and poor efficiency of the NDT thresholding method [62]. The POT technique used in TranAD and other models like OmniAnomaly helps set more accurate threshold values by also considering the localized peak values in the data sequence. DAGMM model performs very well for short datasets like UCR, NAB, MBA and SMAP, but its scores drop significantly for other datasets with longer sequences. This is because it does not map the temporal information explicitly as it does not use sequence windows but only a single GRU model. The window encoder in TranAD, with the encoding of the complete sequence as a self-condition, allows it to perform better even with long high-dimensional sequences. The OmniAnomaly, CAE-M and MSCRED models use sequential observations as input, allowing these methods to retain the temporal information. Such methods perform reconstruction regardless of anomalous data, which prevents them from detecting anomalies close to the normal trends [4]. TranAD tackles this by using adversarial training to amplify errors. Hence, in datasets like SMD, where anomalous data is not very far from normal data, it can detect even mild anomalies.

Recent models such as USAD, MTAD-GAT and GDN use attention mechanisms to focus on specific modes of the data. Moreover, these models try to capture the long-term trends by adjusting the weights of their neural network and only use a local window as an input for reconstruction. GDN has slightly higher scores for MSL and MSDS datasets than TranAD due to the scalable graph-based inference over the inter-dimensional data correlations [14]. TranAD does this using self-attention and performs better than GDN overall across all datasets. The limitation of seeing only a local contextual window prevents methods such as USAD and MTAD-GAT from classifying long-term anomalies (like in SMD or WADI). However, self-conditioning on an embedding of the complete trace with position encoding aids temporal attention, thanks to the transformer

Table 2: Performance comparison of TranAD with baseline methods on the complete dataset. P: Precision, R: Recall, AUC: Area under the ROC curve, F1: F1 score with complete training data. The best F1 and AUC scores are highlighted in bold.

Method	NAB				UCR				MBA			
	P	R	AUC	F1	P	R	AUC	F1	P	R	AUC	F1
MERLIN	0.8013	0.7262	0.8414	0.7619	0.7542	0.8018	0.8984	0.7542	0.9846	0.4913	0.7828	0.6555
LSTM-NDT	0.6400	0.6667	0.8322	0.6531	0.5231	0.8294	0.9781	0.5231	0.9207	0.9718	0.9780	0.9456
DAGMM	0.7622	0.7292	0.8572	0.7453	0.5337	0.9718	0.9916	0.5337	0.9475	0.9900	0.9858	0.9683
OmniAnomaly	0.8421	0.6667	0.8330	0.7442	0.8346	0.9999	0.9981	0.8346	0.8561	1.0000	0.9570	0.9225
MSCRED	0.8522	0.6700	0.8401	0.7502	0.5441	0.9718	0.9920	0.5441	0.9272	1.0000	0.9799	0.9623
MAD-GAN	0.8666	0.7012	0.8478	0.7752	0.8538	0.9891	0.9984	0.8538	0.9396	1.0000	0.9836	0.9689
USAD	0.8421	0.6667	0.8330	0.7442	0.8952	1.0000	0.9989	0.8952	0.8953	0.9989	0.9701	0.9443
MTAD-GAT	0.8421	0.7272	0.8221	0.7804	0.7812	0.9972	0.9978	0.7812	0.9018	1.0000	0.9721	0.9484
CAE-M	0.7918	0.8019	0.8019	0.7968	0.6981	1.0000	0.9957	0.6981	0.8442	0.9997	0.9661	0.9154
GDN	0.8129	0.7872	0.8542	0.7998	0.6894	0.9988	0.9959	0.6894	0.8832	0.9892	0.9528	0.9332
TranAD	0.8889	0.9892	**0.9541**	**0.9364**	0.9407	1.0000	**0.9994**	**0.9407**	0.9569	1.0000	**0.9885**	**0.9780**
Method	SMAP				MSL				SWaT			
	P	R	AUC	F1	P	R	AUC	F1	P	R	AUC	F1
MERLIN	0.1577	0.9999	0.7426	0.2725	0.2613	0.4645	0.6281	0.3345	0.6560	0.2547	0.6175	0.3669
LSTM-NDT	0.8523	0.7326	0.8602	0.7879	0.6288	1.0000	0.9532	0.7721	0.7778	0.5109	0.7140	0.6167
DAGMM	0.8069	0.9891	0.9885	0.8888	0.7363	1.0000	0.9716	0.8482	0.9933	0.6879	0.8436	0.8128
OmniAnomaly	0.8130	0.9419	0.9889	0.8728	0.7848	0.9924	0.9782	0.8765	0.9782	0.6957	0.8467	0.8131
MSCRED	0.8175	0.9216	0.9821	0.8664	0.8912	0.9862	0.9807	0.9363	0.9992	0.6770	0.8433	0.8072
MAD-GAN	0.8157	0.9216	0.9891	0.8654	0.8516	0.9930	0.9862	0.9169	0.9593	0.6957	0.8463	0.8065
USAD	0.7480	0.9627	0.9890	0.8419	0.7949	0.9912	0.9795	0.8822	0.9977	0.6879	0.8460	0.8143
MTAD-GAT	0.7991	0.9991	0.9844	0.8880	0.7917	0.9824	0.9899	0.8768	0.9718	0.6957	0.8464	0.8109
CAE-M	0.8193	0.9567	0.9901	0.8827	0.7751	1.0000	0.9903	0.8733	0.9697	0.6957	0.8464	0.8101
GDN	0.7480	0.9891	0.9864	0.8518	0.9308	0.9892	0.9814	**0.9591**	0.9697	0.6957	0.8462	0.8101
TranAD	0.8043	0.9999	**0.9921**	**0.8915**	0.9038	0.9999	**0.9916**	0.9494	0.9760	0.6997	**0.8491**	**0.8151**
Method	WADI				SMD				MSDS			
	P	R	AUC	F1	P	R	AUC	F1	P	R	AUC	F1
MERLIN	0.0636	0.7669	0.5912	0.1174	0.2871	0.5804	0.7158	0.3842	0.7254	0.3110	0.5022	0.4353
LSTM-NDT	0.0138	0.7823	0.6721	0.0271	0.9736	0.8440	0.9671	0.9042	0.9999	0.8012	0.8013	0.8896
DAGMM	0.0760	0.9981	0.8563	0.1412	0.9103	0.9914	0.9954	0.9491	0.9891	0.8026	0.9013	0.8861
OmniAnomaly	0.3158	0.6541	0.8198	0.4260	0.8881	0.9985	0.9946	0.9401	1.0000	0.7964	0.8982	0.8867
MSCRED	0.2513	0.7319	0.8412	0.3741	0.7276	0.9974	0.9921	0.8414	1.0000	0.7983	0.8943	0.8878
MAD-GAN	0.2233	0.9124	0.8026	0.3588	0.9991	0.8440	0.9933	0.9150	0.9982	0.6107	0.8054	0.7578
USAD	0.1873	0.8296	0.8723	0.3056	0.9060	0.9974	0.9933	0.9495	0.9912	0.7959	0.8979	0.8829
MTAD-GAT	0.2818	0.8012	0.8821	0.4169	0.8210	0.9215	0.9921	0.8683	0.9919	0.7964	0.8982	0.8835
CAE-M	0.2782	0.7918	0.8728	0.4117	0.9082	0.9671	0.9783	0.9367	0.9908	0.8439	0.9013	0.9115
GDN	0.2912	0.7931	0.8777	0.4260	0.7170	0.9974	0.9924	0.8342	0.9989	0.8026	**0.9105**	0.8900
TranAD	0.3529	0.8296	**0.8968**	**0.4951**	0.9262	0.9974	**0.9974**	**0.9605**	0.9999	0.8626	0.9013	**0.9262**

architecture in TranAD. This allows TranAD to capture long-term trends more effectively. Further, due to the meta-learning, TranAD also outperforms baselines with limited training data except for OmniAnomaly on the WADI dataset, indicating its high efficacy even with limited data. OmniAnomaly performs best among all methods on the WADI dataset due to high noise in this dataset and dedicated stochasticity modeling in OmniAnomaly [45]. TranAD is slightly behind this method in terms of F1* and AUC*; however, outperforms it when compared across all datasets and also when given the complete WADI dataset.

We perform critical difference analysis to assess the significance of the differences among the performance of the models. Figure 4 depicts the critical difference diagrams for the F1 and AUC scores based on the Wilcoxon pair-wised signed-rank test (with $\alpha = 0.05$) after rejecting the null hypothesis using the Friedman test on all datasets [22]. TranAD achieves the best rank across all models with a significant statistical difference.

Anomaly Diagnosis. The anomaly diagnosis results in Table 4 where H and N correspond to HitRate and NDCG (with complete data used for model testing). We only present results on the multivariate SMD and MSDS datasets for the sake of brevity (TranAD yields better scores for others as well). We also ignore models that do not explicitly output anomaly class outputs for each dimension individually. Multi-head attention in TranAD allows it to attend to multiple modes simultaneously, making it more suitable for more inter-correlated anomalies. This is observed and explained by datasets like MSDS (distributed systems) where anomalous behavior in one mode can lead to a chain of events causing anomalies

Table 3: Performance comparison of TranAD with baseline methods with 20% of the training dataset. AUC*: AUC with 20% training data, F1*: F1 score with 20% training data. The best F1* and AUC* scores are highlighted in bold.

Method	NAB		UCR		MBA	
	AUC*	F1*	AUC*	F1*	AUC*	F1*
MERLIN	0.8029	0.7619	0.8984	0.7773	0.7828	0.6555
LSTM-NDT	0.8013	0.6212	0.8913	0.5198	0.9617	0.9282
DAGMM	0.7827	0.6125	0.9812	0.5718	0.9671	0.9396
OmniAnomaly	0.8129	0.6713	0.9728	0.7918	0.9407	0.9217
MSCRED	0.8299	0.7013	0.9637	0.4929	0.9499	0.9108
MAD-GAN	0.8194	0.7109	0.9959	0.8216	0.9550	0.9192
USAD	0.7267	0.6781	0.9967	0.8538	0.9697	0.9425
MTAD-GAT	0.6956	0.7013	0.9974	0.8671	0.9688	0.9425
CAE-M	0.7312	0.7126	0.9926	0.7525	0.9616	0.9002
GDN	0.8299	0.7013	0.9937	0.8029	0.9671	0.9316
TranAD	**0.9217**	**0.8421**	**0.9989**	**0.9399**	**0.9718**	**0.9617**

Method	SMAP		MSL		SWaT	
	AUC*	F1*	AUC*	F1*	AUC*	F1*
MERLIN	0.7426	0.2725	0.6281	0.3345	0.6175	0.3669
LSTM-NDT	0.7007	0.5418	0.9520	0.7608	0.6690	0.4145
DAGMM	0.9881	0.8369	0.9606	0.8010	0.8421	0.8001
OmniAnomaly	0.9879	0.8131	0.9703	0.8424	0.8319	0.7433
MSCRED	0.9811	0.8050	0.9797	0.8232	0.8385	0.7922
MAD-GAN	0.9877	0.8468	0.9649	0.8190	0.8456	0.8012
USAD	0.9883	0.8379	0.9649	0.8190	0.8438	0.8087
MTAD-GAT	0.9814	0.8225	0.9782	0.8024	**0.8459**	0.8079
CAE-M	**0.9892**	0.8312	0.9836	0.7303	0.8458	0.7841
GDN	0.9887	0.8411	0.9414	0.8959	0.8390	0.8072
TranAD	0.9885	**0.8889**	**0.9857**	**0.9172**	0.8438	**0.8094**

Method	WADI		SMD		MSDS	
	AUC*	F1*	AUC*	F1*	AUC*	F1*
MERLIN	0.5912	0.1174	0.7158	0.3842	0.5022	0.4353
LSTM-NDT	0.6637	0.0000	0.9563	0.6754	0.7813	0.7912
DAGMM	0.6497	0.0630	0.9845	0.8986	0.7763	0.8389
OmniAnomaly	**0.7913**	**0.1017**	0.9859	0.9352	0.5613	0.8389
MSCRED	0.6029	0.0413	0.9768	0.8004	0.7716	0.8283
MAD-GAN	0.5383	0.0937	0.8635	0.9318	0.5002	0.7390
USAD	0.7011	0.0733	0.9854	0.9213	0.7613	0.8389
MTAD-GAT	0.6267	0.0520	0.9798	0.6661	0.6122	0.8248
CAE-M	0.6109	0.0781	0.9569	0.9318	0.6001	0.8389
GDN	0.6121	0.0412	0.9811	0.7107	0.6819	0.8389
TranAD	0.7688	0.0649	**0.9869**	**0.9478**	**0.8113**	**0.8391**

in other modes (see Figure 5). TranAD is able to leverage the complete trace information with the local window to aid in pinpointing anomalous behavior to specific modes. The table demonstrates that TranAD is able to detect 46.3% – 75.3% root causes for anomalies.

Compared to the baseline methods, TranAD is able to improve diagnosis score by up to 6% for SMD and 30% for MSDS. The average improvement in diagnosis scores is 4.25%.

5 ANALYSES

5.1 Ablation Analysis

To study the relative importance of each component of the model, we exclude every major one and observe how it affects the performance in terms of the F1 scores for each dataset. First, we consider the TranAD model without the transformer-based encoder-decoder

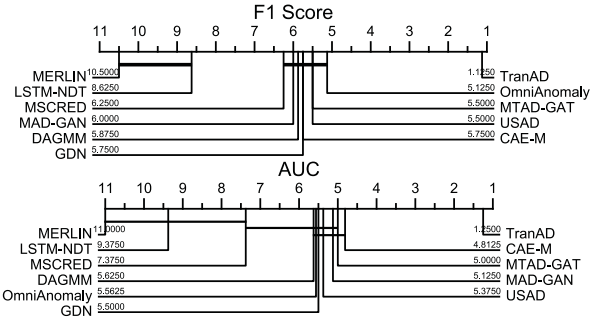


Figure 4: Critical difference diagrams for F1 and AUC scores using the Wilkoxon pairwised signed rank test (with $\alpha = 0.05$) on all datasets. Rightmost methods are ranked higher.

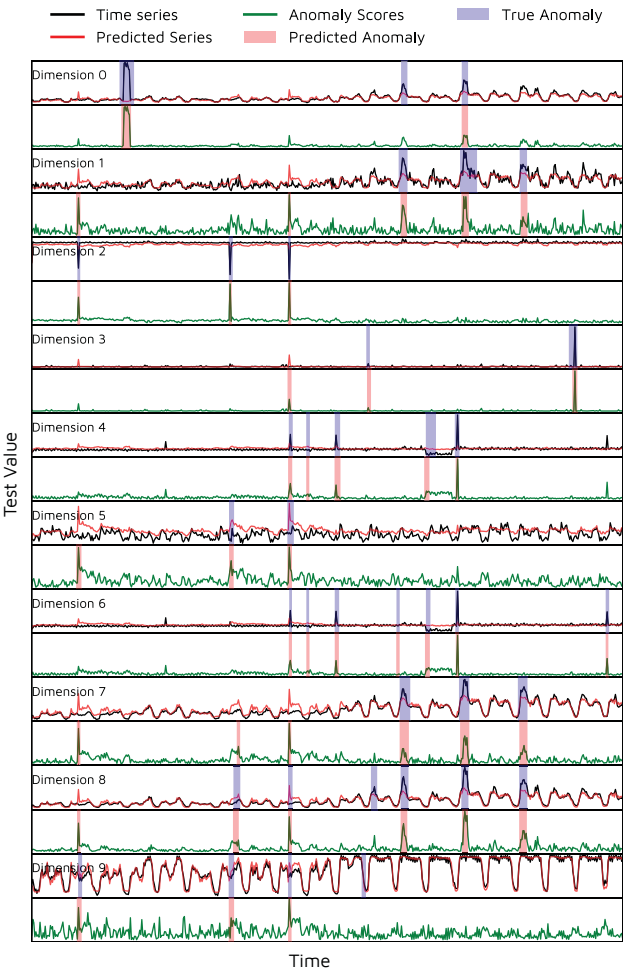


Figure 5: Predicted and Ground Truth labels for the MSDS test set using the TranAD model.

architecture but instead with a feed-forward network. Second, we consider the model without the self-conditioning, *i.e.*, we fix the focus score $F = \vec{0}$ in phase 2. Third, we study the model without

Table 4: Diagnosis Performance.

Method	SMD				MSDS			
	H@100%	H@150%	N@100%	N@150%	H@100%	H@150%	N@100%	N@150%
MERLIN	0.5907	0.6177	0.4150	0.4912	0.3816	0.5626	0.3010	0.3947
LSTM-NDT	0.3808	0.5225	0.3603	0.4451	0.1504	0.2959	0.1124	0.1993
DAGMM	0.4927	0.6091	0.5169	0.5845	0.2617	0.4333	0.3153	0.4154
OmniAnomaly	0.4567	0.5652	0.4545	0.5125	0.2839	0.4365	0.3338	0.4231
MSCRED	0.4272	0.5180	0.4609	0.5164	0.2322	0.3469	0.2297	0.2962
MAD-GAN	0.4630	0.5785	0.4681	0.5522	0.3856	0.5589	0.4277	0.5292
USAD	0.4925	0.6055	0.5179	0.5781	0.3095	0.4769	0.3534	0.4515
MTAD-GAT	0.3493	0.4777	0.3759	0.4530	**0.5812**	0.5885	0.5926	0.6522
CAE-M	0.4707	0.5878	**0.5474**	**0.6178**	0.2530	0.4171	0.2047	0.3010
GDN	0.3143	0.4386	0.2980	0.3724	0.2276	0.3382	0.2921	0.3570
TranAD	**0.4981**	**0.6401**	0.4941	**0.6178**	0.4630	**0.7533**	**0.5981**	**0.6963**

Table 5: Comparison of training times in seconds per epoch.

Method	NAB	UCR	MBA	SMAP	MSL	SWaT	WADI	SMD	MSDS
MERLIN	3.28	4.09	20.19	6.89	5.12	10.12	132.69	72.32	42.22
LSTM-NDT	10.64	8.71	27.80	27.62	26.24	26.43	297.12	373.14	361.12
DAGMM	25.38	20.78	74.62	19.05	16.41	18.51	178.17	204.36	187.54
OmniAnomaly	38.27	27.96	109.86	27.05	21.31	28.39	212.99	276.97	277.10
MSCRED	258.86	262.45	592.13	16.13	33.47	183.67	1349.05	237.66	109.63
MAD-GAN	39.80	25.71	160.29	29.49	26.27	27.79	293.60	314.82	285.25
USAD	31.21	21.10	120.86	23.63	21.22	22.72	242.86	250.97	232.82
MTAD-GAT	145.00	97.12	233.08	1015.03	1287.42	103.92	9812.13	6564.11	1304.09
CAE-M	22.48	19.42	67.44	187.35	575.96	41.25	5525.62	3102.12	552.83
GDN	83.84	58.78	159.01	62.33	96.71	59.40	4063.05	809.94	585.34
TranAD	**1.25**	**0.84**	**4.08**	**3.55**	**5.27**	**0.87**	**115.91**	**43.56**	**17.15**

Table 6: Ablation Study - F1 and F1* scores for TranAD and its ablated versions.

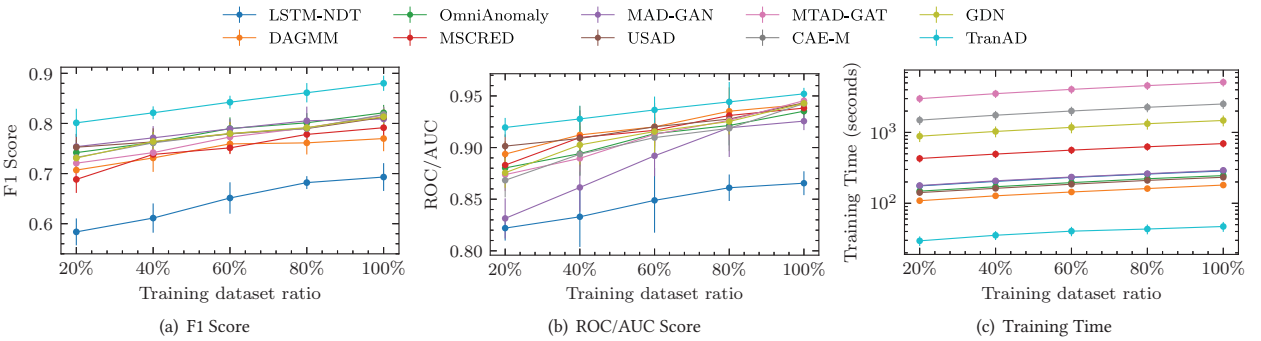
Method	NAB		UCR		MBA	
	F1	F1*	F1	F1*	F1	F1*
TranAD	**0.9364**	**0.8421**	**0.9694**	**0.9399**	**0.9780**	**0.9617**
w/o transformer	0.8850	0.8019	0.8466	0.5495	0.9749	0.9584
w/o self-condition	0.8887	0.8102	0.9191	0.9028	0.9770	**0.9617**
w/o adversarial training	0.9012	0.8102	0.9634	0.9289	0.9752	0.9592
w/o MAML	0.9068	0.8210	0.9689	0.9304	0.9756	**0.9617**
Method	SMAP		MSL		SWaT	
	F1	F1*	F1	F1*	F1	F1*
TranAD	**0.8915**	**0.8889**	**0.9494**	**0.9172**	**0.8151**	**0.8094**
w/o transformer	0.8643	0.8147	0.9137	0.9037	0.8143	0.6360
w/o self-condition	0.8894	0.8153	0.9175	0.8913	0.7953	0.8094
w/o adversarial training	0.8906	0.8476	0.9455	0.9172	0.8028	0.7832
w/o MAML	**0.8915**	**0.8899**	0.9466	0.6732	0.8143	0.8079
Method	WADI		SMD		MSDS	
	F1	F1*	F1	F1*	F1	F1*
TranAD	**0.4951**	**0.0649**	**0.9605**	**0.9478**	**0.9262**	**0.8391**
w/o transformer	0.2181	0.0037	0.9071	0.9032	0.8867	0.8389
w/o self-condition	0.3620	0.0631	0.9502	0.8847	0.8748	0.8214
w/o adversarial training	0.3820	0.0621	0.9177	0.8667	0.9181	0.8389
w/o MAML	0.4815	0.0553	0.9433	0.8164	0.8870	0.8389

the adversarial loss, *i.e.*, a single-phase inference and only the reconstruction loss for model training. Finally, we consider the model without meta-learning. The results are summarized in Table 6 and provide the following findings:

- Replacing the transformer-based encoder-decoder has the highest performance drop of nearly 11% in terms of the F1 score. This drop is more pronounced for the WADI dataset (56%), demonstrating the need for the attention-based transformer for large-scale datasets.
- When we remove the self-conditioning, the average drop in F1 scores is 6%, which shows that the focus score aids prediction performance.
- Removing the two-phase adversarial training mainly affects SMD and WADI datasets as these traces have a large proportion of mild anomalies and the adversarial loss helps amplify the anomaly scores. The average drop in F1 score, in this case, is 5%.
- Not having the meta-learning in the model has little effect to the F1 scores ($\approx$1%); however, it leads to a nearly 12% drop in F1*.

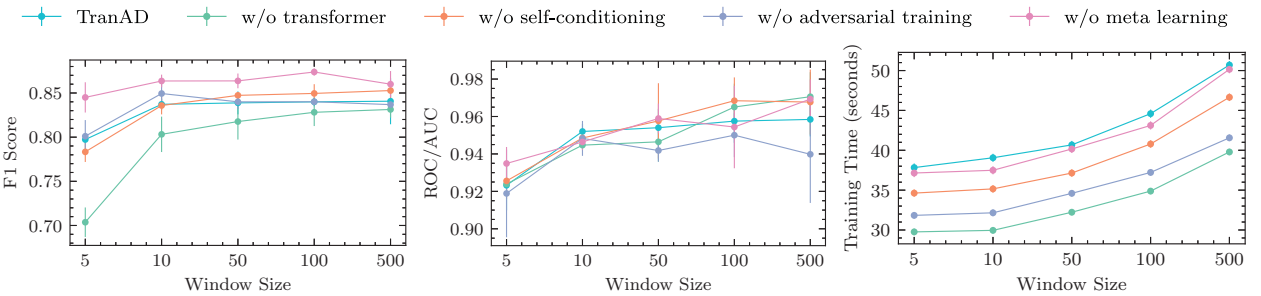
5.2 Overhead Analysis

Table 5 shows the average training times for all models on every dataset in seconds per epoch. For comparison, we report the training time for MERLIN as the time it takes to discover discords in



(a) F1 Score (b) ROC/AUC Score (c) Training Time

Figure 6: F1 score, ROC/AUC score and training times with dataset size.



(a) F1 Score (b) ROC/AUC Score (c) Training Time

Figure 7: F1 score, ROC/AUC score and training times with window size.

the test data. The training time of TranAD is 75% – 99% lower than those of the baselines. This demonstrates the advantage of having a transformer with positional encoding to push the complete sequence as an input instead of inferring over sequential windows.

5.3 Sensitivity Analysis

Sensitivity to the training set size. Figure 6 shows the variation of the F1 and AUC scores of all models averaged for all datasets, and the training time with the ratio of the training data used for model training, ranging from 20% to 100%. We do not report sensitivity results on MERLIN as it does not use training data. Other deep learning based reconstruction models are given the same randomly sampled subsequence of 20% to 100% size as that of the training data. We report 90% confidence bounds in Figure 6. Clearly, as dataset size increases, the prediction performance improves and the training time increases. We observe that for every ratio, the TranAD model has a higher F1 score and a lower training time.

Sensitivity to the window size. We also show the performance of the TranAD model and its variants with different window sizes in Figure 7. The window size has an impact both on the anomaly detection scores and training times. TranAD can detect anomalies faster when we have smaller windows since the inference time is lower for smaller inputs. If the window is too small, it does not represent the local contextual information well. However, if the window is too large, short anomalies may be hidden in a large number of datapoints in such a window (see the drop in F1 score for some models). A window size of 10 gives a reasonable balance between the F1 score and training times and hence is used in our experiments.

6 CONCLUSIONS

We present a transformer based anomaly detection model (TranAD) that can detect and diagnose anomalies for multivariate time-series data. The transformer based encoder-decoder allows quick model training and high detection performance for a variety of datasets considered in this work. TranAD leverages self-conditioning and adversarial training to amplify errors and gain training stability. Moreover, meta-learning allows it to be able to identify data trends even with limited data. Specifically, TranAD achieves an improvement of 17% and 11% for F1 score on complete and limited training data, respectively. It is also able to correctly identify root causes for up to 75% of the detected anomalies, higher than the state-of-the-art models. It is able to achieve this with up to 99% lower training times compared to the baseline methods. This makes TranAD an ideal choice for modern industrial systems where accurate and quick anomaly predictions are required.

For the future, we propose to extend the method with other transformer models like bidirectional neural networks to allow model generalization to diverse temporal trends in data. We also wish to explore the direction of applying cost-benefit analysis for each model component based on the deployment setting to avoid expensive computation.

REFERENCES

[1] Hossein Abbasimehr, Mostafa Shabani, and Mohsen Yousefi. 2020. An optimized model using LSTM network for demand forecasting. *Computers & industrial engineering* 143 (2020), 106435.

[2] Subutai Ahmad, Alexander Lavin, Scott Purdy, and Zuha Agha. 2017. Unsupervised real-time anomaly detection for streaming data. *Neurocomputing* 262 (2017), 134–147.

[3] Chuadhry Mujeeb Ahmed, Venkata Reddy Palleti, and Aditya P Mathur. 2017. WADI: a water distribution testbed for research in the design of secure cyber physical systems. In *Proceedings of the 3rd International Workshop on Cyber-Physical Systems for Smart Water Networks*. 25–28.

[4] Julien Audibert, Pietro Michiardi, Frédéric Guyard, Sébastien Marti, and Maria A Zuluaga. 2020. USAD: UnSupervised Anomaly Detection on Multivariate Time Series. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 3395–3404.

[5] Tharindu R Bandaragoda, Kai Ming Ting, David Albrecht, Fei Tony Liu, and Jonathan R Wells. 2014. Efficient anomaly detection by isolation using nearest neighbour ensemble. In *2014 IEEE International Conference on Data Mining Workshop*. IEEE, 698–705.

[6] Julian Bellendorf and Zoltán Ádám Mann. 2020. Classification of optimization problems in fog computing. *Future Generation Computer Systems* 107 (2020), 158–176.

[7] Nejc Bezak, Mitja Brilly, and Mojca Šraj. 2014. Comparison between the peaks-over-threshold method and the annual maximum method for flood frequency analysis. *Hydrological Sciences Journal* 59, 5 (2014), 959–977.

[8] Paul Boniol, Michele Linardi, Federico Roncallo, and Themis Palpanas. 2020. Automated anomaly detection in large sequences. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1834–1837.

[9] Paul Boniol, Themis Palpanas, Mohammed Meftah, and Emmanuel Remy. 2020. Graphan: Graph-based subsequence anomaly detection. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2941–2944.

[10] Paul Boniol, John Paparrizos, Themis Palpanas, and Michael J. Franklin. 2021. SAND: Streaming Subsequence Anomaly Detection. *Proc. VLDB Endow.* 14, 10 (2021), 1717–1729.

[11] Saikiran Bulusu, Bhavya Kailkhura, Bo Li, Pramod K Varshney, and Dawn Song. 2020. Anomalous example detection in deep learning: A survey. *IEEE Access* 8 (2020), 132330–132347.

[12] Raghavendra Chalapathy and Sanjay Chawla. 2019. Deep learning for anomaly detection: A survey. *arXiv preprint arXiv:1901.03407* (2019).

[13] Hoang Anh Dau, Eamonn Keogh, Kaveh Kamgar, Chin-Chia Michael Yeh, Yan Zhu, Shaghayegh Gharghabi, Chotirat Ann Ratanamahatana, Yanping, Bing Hu, Nurjahan Begum, Anthony Bagnall, Abdullah Mueen, Gustavo Batista, and Hexagon-ML. 2018. The UCR Time Series Classification Archive. `https://www.cs.ucr.edu/~eamonn/time_series_data_2018/`.

[14] Ailin Deng and Bryan Hooi. 2021. Graph neural network-based anomaly detection in multivariate time series. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35. 4027–4035.

[15] Chelsea Finn, Pieter Abbeel, and Sergey Levine. 2017. Model-agnostic meta-learning for fast adaptation of deep networks. In *International Conference on Machine Learning*. PMLR, 1126–1135.

[16] Shaghayegh Gharghabi, Shima Imani, Anthony Bagnall, Amirali Darvishzadeh, and Eamonn Keogh. 2018. Matrix profile XII: MPDIST: a novel time series distance measure to allow data mining in more challenging scenarios. In *2018 IEEE International Conference on Data Mining (ICDM)*. IEEE, 965–970.

[17] Ary L Goldberger, Luis AN Amaral, Leon Glass, Jeffrey M Hausdorff, Plamen Ch Ivanov, Roger G Mark, Joseph E Mietus, George B Moody, Chung-Kang Peng, and H Eugene Stanley. 2000. PhysioBank, PhysioToolkit, and PhysioNet: components of a new research resource for complex physiologic signals. *circulation* 101, 23 (2000), e215–e220.

[18] Xin He, Kaiyong Zhao, and Xiaowen Chu. 2021. AutoML: A Survey of the State-of-the-Art. *Knowledge-Based Systems* 212 (2021), 106622.

[19] Shaohan Huang, Yi Liu, Carol Fung, Rong He, Yining Zhao, Hailong Yang, and Zhongzhi Luan. 2020. HitAnomaly: Hierarchical Transformers for Anomaly Detection in System Log. *IEEE Transactions on Network and Service Management* 17, 4 (2020), 2064–2076.

[20] Kyle Hundman, Valentino Constantinou, Christopher Laporte, Ian Colwell, and Tom Soderstrom. 2018. Detecting spacecraft anomalies using LSTMs and nonparametric dynamic thresholding. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*. 387–395.

[21] Shima Imani, Frank Madrid, Wei Ding, Scott Crouter, and Eamonn Keogh. 2018. Matrix profile xiii: Time series snippets: a new primitive for time series data mining. In *2018 IEEE international conference on big knowledge (ICBK)*. IEEE, 382–389.

[22] Hassan Ismail Fawaz, Germain Forestier, Jonathan Weber, Lhassane Idoumghar, and Pierre-Alain Muller. 2019. Deep learning for time series classification: a review. *Data Mining and Knowledge Discovery* 33, 4 (2019), 917–963.

[23] Vincent Jacob, Fei Song, Arnaud Stiegler, Bijan Rad, Yanlei Diao, and Nesime Tatbul. 2020. Exathlon: A Benchmark for Explainable Anomaly Detection over Time Series. *Proceedings of the VLDB Endowment* (2020).

[24] Kalervo Järvelin and Jaana Kekäläinen. 2002. Cumulated gain-based evaluation of IR techniques. *ACM Transactions on Information Systems (TOIS)* 20, 4 (2002), 422–446.

[25] Stratis Kanarachos, Jino Mathew, Alexander Chroneos, and M Fitzpatrick. 2015. Anomaly detection in time series data using a combination of wavelets, neural networks and Hilbert transform. In *2015 6th International Conference on Information, Intelligence, Systems and Applications (IISA)*. IEEE, 1–6.

[26] Eamonn Keogh, Dutta Roy Taposh, U Naik, and A Agrawal. 2021. Multi-dataset Time-Series Anomaly Detection Competition. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. `https://compete.hexagon-ml.com/practice/competition/39/`.

[27] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[28] Kyle Kingsbury and Peter Alvaro. 2020. Elle: inferring isolation anomalies from experimental observations. *Proceedings of the VLDB Endowment* 14, 3 (2020), 268–280.

[29] Dan Li, Dacheng Chen, Baihong Jin, Lei Shi, Jonathan Goh, and See-Kiong Ng. 2019. MAD-GAN: Multivariate anomaly detection for time series data with generative adversarial networks. In *International Conference on Artificial Neural Networks*. Springer, 703–716.

[30] Guoliang Li, Xuanhe Zhou, Ji Sun, Xiang Yu, Yue Han, Lianyuan Jin, Wenbo Li, Tianqing Wang, and Shifu Li. 2021. opengauss: An autonomous database system. *Proceedings of the VLDB Endowment* 14, 12 (2021), 3028–3042.

[31] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. 2008. Isolation forest. In *2008 eighth ieee international conference on data mining*. IEEE, 413–422.

[32] Steven Liu, Tongzhou Wang, David Bau, Jun-Yan Zhu, and Antonio Torralba. 2020. Diverse image generation via self-conditioned GANs. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 14286–14295.

[33] Aditya P Mathur and Nils Ole Tippenhauer. 2016. SWaT: a water treatment testbed for research and training on ICS security. In *2016 international workshop on cyber-physical systems for smart water networks (CySWater)*. IEEE, 31–36.

[34] Gideon Mbiydzenyuy. 2020. Univariate Time Series Anomaly Labelling Algorithm. In *International Conference on Machine Learning, Optimization, and Data Science*. Springer, 586–599.

[35] Steena Monteiro, Forrest Iandola, and Daniel Wong. 2016. STOMP: Statistical Techniques for Optimizing and Modeling Performance of blocked sparse matrix vector multiplication. In *2016 28th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, 93–100.

[36] George B Moody and Roger G Mark. 2001. The impact of the MIT-BIH arrhythmia database. *IEEE Engineering in Medicine and Biology Magazine* 20, 3 (2001), 45–50.

[37] Takaaki Nakamura, Makoto Imamura, Ryan Mercer, and Eamonn Keogh. 2020. MERLIN: Parameter-Free Discovery of Arbitrary Length Anomalies in Massive Time Series Archives. In *2020 IEEE International Conference on Data Mining (ICDM)*. IEEE, 1190–1195.

[38] Sasho Nedelkoski, Jasmin Bogatinovski, Ajay Kumar Mandapati, Soeren Becker, Jorge Cardoso, and Odej Kao. 2020. Multi-source distributed system data for AI-powered analytics. In *European Conference on Service-Oriented and Cloud Computing*. Springer, 161–176.

[39] Daehyung Park, Yuuna Hoshi, and Charles C Kemp. 2018. A multimodal anomaly detector for robot-assisted feeding using an LSTM-based variational autoencoder. *IEEE Robotics and Automation Letters* 3, 3 (2018), 1544–1551.

[40] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *Advances in Neural Information Processing Systems* 32 (2019), 8026–8037.

[41] Animesh Patcha and Jung-Min Park. 2007. An overview of anomaly detection techniques: Existing solutions and latest technological trends. *Computer networks* 51, 12 (2007), 3448–3470.

[42] Noorhan Saleh and Maggie Mashaly. 2019. A Dynamic Simulation Environment for Container-based Cloud Data Centers using ContainerCloudSim. In *2019 Ninth International Conference on Intelligent Computing and Information Systems (ICICIS)*. IEEE, 332–336.

[43] Osman Salem, Alexey Guerassimov, Ahmed Mehaoua, Anthony Marcus, and Borko Furht. 2014. Anomaly detection in medical wireless sensor networks using SVM and linear regression models. *International Journal of E-Health and Medical Communications (IJEHMC)* 5, 1 (2014), 20–45.

[44] Alban Siffer, Pierre-Alain Fouque, Alexandre Termier, and Christine Largouet. 2017. Anomaly detection in streams with extreme value theory. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 1067–1075.

[45] Ya Su, Youjian Zhao, Chenhao Niu, Rong Liu, Wei Sun, and Dan Pei. 2019. Robust anomaly detection for multivariate time series through stochastic recurrent neural network. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2828–2837.

[46] Srikanth Thudumu, Philip Branch, Jiong Jin, and Jugdutt Jack Singh. 2020. A comprehensive survey of anomaly detection techniques for high dimensional big data. *Journal of Big Data* 7, 1 (2020), 1–30.

[47] Luan Tran, Min Y Mun, and Cyrus Shahabi. 2020. Real-time distance-based outlier detection in data streams. *Proceedings of the VLDB Endowment* 14, 2 (2020), 141–153.

[48] Shreshth Tuli, Giuliano Casale, and Nicholas R Jennings. 2022. PreGAN: Preemptive Migration Prediction Network for Proactive Fault-Tolerant Edge Computing. In *IEEE Conference on Computer Communications (INFOCOM)*. IEEE.

[49] Shreshth Tuli, Giuliano Casale, and Nicholas R Jennings. 2022. TranAD: Deep Transformer Networks for Anomaly Detection in Multivariate Time Series Data. *arXiv preprint arXiv:2201.07284* (2022).

[50] Shreshth Tuli, Shivananda Poojara, Satish Narayana Srirama, Giuliano Casale, and Nick Jennings. 2021. COSCO: Container Orchestration using Co-Simulation and Gradient Based Optimization for Fog Computing Environments. *IEEE Transactions on Parallel and Distributed Systems* (2021).

[51] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*. 6000–6010.

[52] Yiyang Wang, Neda Masoud, and Anahita Khojandi. 2020. Real-time sensor anomaly detection and recovery in connected automated vehicle sensors. *IEEE Transactions on Intelligent Transportation Systems* 22, 3 (2020), 1411–1421.

[53] Y Webscope. [n.d.]. S5-A Labeled Anomaly Detection Dataset, Version 1.0. `https://webscope.sandbox.yahoo.com/catalog.php?datatype=s&did=70`. Accessed: 2021-08-31.

[54] Krzysztof Witkowski. 2017. Internet of things, big data, industry 4.0–innovative solutions in logistics and supply chains management. *Procedia engineering* 182 (2017), 763–769.

[55] Renjie Wu and Eamonn J Keogh. 2020. Current Time Series Anomaly Detection Benchmarks are Flawed and are Creating the Illusion of Progress. *arXiv preprint arXiv:2009.13807* (2020).

[56] Asrul H Yaacob, Ian KT Tan, Su Fong Chien, and Hon Khi Tan. 2010. Arima based network anomaly detection. In *2010 Second International Conference on Communication Software and Networks*. IEEE, 205–209.

[57] Qiang Yang, Yang Liu, Tianjian Chen, and Yongxin Tong. 2019. Federated machine learning: Concept and applications. *ACM Transactions on Intelligent Systems and Technology (TIST)* 10, 2 (2019), 1–19.

[58] Dragomir Yankov, Eamonn Keogh, and Umaa Rebbapragada. 2008. Disk aware discord discovery: Finding unusual time series in terabyte sized datasets. *Knowledge and Information Systems* 17, 2 (2008), 241–262.

[59] Chin-Chia Michael Yeh, Yan Zhu, Liudmila Ulanova, Nurjahan Begum, Yifei Ding, Hoang Anh Dau, Diego Furtado Silva, Abdullah Mueen, and Eamonn Keogh. 2016. Matrix profile I: all pairs similarity joins for time series: a unifying view that includes motifs, discords and shapelets. In *2016 IEEE 16th international conference on data mining (ICDM)*. Ieee, 1317–1322.

[60] Chuxu Zhang, Dongjin Song, Yuncong Chen, Xinyang Feng, Cristian Lumezanu, Wei Cheng, Jingchao Ni, Bo Zong, Haifeng Chen, and Nitesh V Chawla. 2019. A deep neural network for unsupervised anomaly detection and diagnosis in multivariate time series data. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 1409–1416.

[61] Yuxin Zhang, Yiqiang Chen, Jindong Wang, and Zhiwen Pan. 2021. Unsupervised Deep Anomaly Detection for Multi-Sensor Time-Series Signals. *IEEE Transactions on Knowledge and Data Engineering* (2021).

[62] Hang Zhao, Yujing Wang, Juanyong Duan, Congrui Huang, Defu Cao, Yunhai Tong, Bixiong Xu, Jing Bai, Jie Tong, and Qi Zhang. 2020. Multivariate time-series anomaly detection via graph attention network. *International Conference on Data Mining* (2020).

[63] Yan Zhu, Chin-Chia Michael Yeh, Zachary Zimmerman, Kaveh Kamgar, and Eamonn Keogh. 2018. Matrix profile XI: SCRIMP++: time series motif discovery at interactive speeds. In *2018 IEEE International Conference on Data Mining (ICDM)*. IEEE, 837–846.

[64] Zachary Zimmerman, Nader Shakibay Senobari, Gareth Funning, Evangelos Papalexakis, Samet Oymak, Philip Brisk, and Eamonn Keogh. 2019. Matrix profile XVIII: time series mining in the face of fast moving streams using a learned approximate matrix profile. In *2019 IEEE International Conference on Data Mining (ICDM)*. IEEE, 936–945.

[65] Bo Zong, Qi Song, Martin Renqiang Min, Wei Cheng, Cristian Lumezanu, Daeki Cho, and Haifeng Chen. 2018. Deep autoencoding gaussian mixture model for unsupervised anomaly detection. In *International Conference on Learning Representations*.

SpaceSaving$^{\pm}$: An Optimal Algorithm for Frequency Estimation and Frequent Items in the Bounded-Deletion Model

Fuheng Zhao
UC Santa Barbara
fuheng_zhao@ucsb.edu

Divyakant Agrawal
UC Santa Barbara
agrawal@cs.ucsb.edu

Amr El Abbadi
UC Santa Barbara
amr@cs.ucsb.edu

Ahmed Metwally
Uber, Inc.
ametwally@uber.com

ABSTRACT

In this paper, we propose the first deterministic algorithms to solve the frequency estimation and frequent item problems in the *bounded-deletion* model. We establish the space lower bound for solving the deterministic frequent items problem in the bounded-deletion model, and propose Lazy SpaceSaving$^{\pm}$ and SpaceSaving$^{\pm}$ algorithms with optimal space bound. We develop an efficient implementation of the SpaceSaving$^{\pm}$ algorithm that minimizes the latency of update operations using novel data structures. The experimental evaluations testify that SpaceSaving$^{\pm}$ has accurate frequency estimations and achieves very high recall and precision across different data distributions while using minimal space. Our experiments clearly demonstrate that, if allowed the same space, SpaceSaving± is more accurate than the state-of-the-art protocols with up to $\frac{logU-1}{logU}$ of the items deleted, where U is the size of the input universe. Moreover, motivated by prior work, we propose Dyadic SpaceSaving$^{\pm}$, the first deterministic quantile approximation sketch in the bounded-deletion model.

PVLDB Reference Format:
Fuheng Zhao, Divyakant Agrawal, Amr El Abbadi, and Ahmed Metwally. SpaceSaving$^{\pm}$: An Optimal Algorithm for Frequency Estimation and Frequent Items in the Bounded-Deletion Model . PVLDB, 15(6): 1215 - 1227, 2022.
doi:10.14778/3514061.3514068

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/ZhaoFuheng/SpaceSavingBoundedDeletionModel.

1 INTRODUCTION

With the development of new technologies and advancements in digital devices, massive amounts of data are generated each day and these data contain crucial information that needs to be analyzed. To make the best use of streaming big data, data sketch[1] algorithms are often leveraged to process the data only once and to provide essential analysis and statistical measures with strong accuracy guarantees while using limited resources. For instance, with limited space and one pass on the dataset, Hyperloglog [23] enables cardinality estimation, the Bloom Filter [8] answers set membership, and KLL [28, 33] provides quantile approximation.

Two fundamental problems in data streams are identifying the most frequently occurring items, a.k.a. frequent items, heavy hitters, Top-K, elephants, iceberg queries, and estimating the frequency of an item, a.k.a the frequency estimation problem. The formal definition of these two problems are included in Section 2.1. Several algorithms [12, 17, 36, 38] have been proposed to solve both problems with tunable accuracy guarantees using small memory footprints. These algorithms can be categorized into *counter* based and *linear sketch* based approaches. The counter based approach [38] tracks a subset of input items and their estimated frequencies. The linear sketch based approach [12, 17, 30] tracks attribute information from the universe. While linear sketches [12, 17] directly solve the frequency estimation problem, they require additional structures such as heaps or need to impose hierarchical structures over the assumed-bounded universe to solve the frequent items problem. The frequency estimation and frequent items problems have important applications, such as click stream analysis [20, 26, 38], distributed caches [45], database management [14, 22, 40, 43], and network monitoring [5, 27, 42]. In addition, if inputs are drawn from a bounded universe, frequency estimation sketches can also solve the quantile approximation problem [17, 44].

Historically, all sketches assumed the *insertion-only* model or the *turnstile* model. The insertion-only model consists only of insert operations, whereas the turnstile model consists of both insert and delete operations such that deletes are always performed on previously inserted items [44]. Supporting both insert and delete operations is harder, e.g., sketches in the turnstile model incur larger space overhead and higher update times compared to sketches in the insertion-only model. Jayaram et al. [29] observed that in practice many turnstile models only incur a fraction of deletions and proposed an intermediate model, the *bounded-deletions* model, in which at most $(1-\frac{1}{\alpha})$ of prior insertions are deleted where $\alpha \geq 1$ and $(1-\frac{1}{\alpha})$ upper bounds the delete:insert ratio. Setting α to 1, the bounded-deletion model becomes the insertion-only model.

The bounded-deletion model is important in many real-world applications such as summarizing product sales in electronic commerce platforms and rankings in standardized testing. Many companies use purchase frequency to check if their customers are satisfied with a product and to identify important groups for advertising

Proceedings of the VLDB Endowment, Vol. 15, No. 6 ISSN 2150-8097.
doi:10.14778/3514061.3514068

[1]The term sketch refers to the algorithms and data structures that can extract valuable information through **one pass** on the entire data.

and marketing campaigns. After customers purchase products, a certain percentage of the purchases may be returned and the frequency estimation should reflect these changes. However, for any financially viable company, it is highly unlikely that all of these customers will return their purchases and hence in most cases the bounded-deletion model can be assumed. In the context of standardized testing such as SAT, ACT, and GRE, frequency estimations are often used to compare and contrast performance among different demographics[2]. Students may request regrades of their exams only once to rectify any machine errors or human errors. Hence, the bounded-deletion model can be used with $\alpha = 2$. Recently, the bounded-deletion model has gained in popularity, and several algorithms have been proposed to discover novel properties of streaming tasks [10, 29, 32, 48].

In this paper, we present the SpaceSaving$^{\pm}$ algorithm that solves both frequency estimation and frequent items problems in the bounded-deletion model with state-of-the-art performance and minimal memory footprint. If the administrator of a large data set knows, a priori, that deletions are not arbitrarily frequent compared to insertions, then SpaceSaving$^{\pm}$ can efficiently capture these changes and identify frequent items with small space, fast update time, and high accuracy. In addition, inspired by quantile summaries [17, 24, 44], we further demonstrate how to leverage SpaceSaving$^{\pm}$ to support deterministic quantile approximation in the bounded-deletion model. In summary, the main contributions of this paper are: (i) we present Lazy SpaceSaving$^{\pm}$ and SpaceSaving$^{\pm}$, two space optimal deterministic algorithms in the bounded-deletion model and establish their space optimality and correctness; (ii) we propose the Dyadic SpaceSaving$^{\pm}$ sketch, the first deterministic quantile approximation sketch in the bounded-deletion model; (iii) we implement SpaceSaving$^{\pm}$ using two heaps to minimize the update time; and (iv) we evaluate SpaceSaving$^{\pm}$ and compare it to the state-of-the-art approaches [12, 17, 29] and achieve 5 orders of magnitude better accuracy on a real-world dataset.

The paper is organized as follows, Section 2 discusses the background of frequency estimation and frequent items problem, and gives an overview of previous algorithms. Section 3 introduces Lazy SpaceSaving$^{\pm}$ and SpaceSaving$^{\pm}$ in the bounded-deletion model, demonstrates that these algorithms are space optimal, and presents an efficient implementation using a min heap and a max heap data structure to minimize update time. Section 4 shows the experimental evaluations conducted using synthetic and real world datasets and compares SpaceSaving$^{\pm}$ to the state-of-the-art sketches that support delete operations. Section 5 introduces the Dyadic SpaceSaving$^{\pm}$ quantile sketch to solve the deterministic quantile approximation problem in the bounded-deletion model. Finally, Section 6 summarizes our contributions and concludes this work.

2 BACKGROUND

2.1 Preliminaries

Given a stream $\sigma = \{item_t\}_{t\in\{1,2,...,N\}}$ of length N and items drawn from *universe* of size U, the frequency of an item x is $f(x) = \sum_{t=1}^{N} \pi(item_t = x)$ where π returns 1 if $item_t$ is x, and returns 0 otherwise. The stream σ implicitly defines a frequency vector $F = \{f(a_1), ..., f(a_U)\}$ for items a_1,...,a_U in the *universe*. Some algorithms assume the *universe* is bounded, such as in the IP network monitoring context [42]. Many algorithms assume unit updates, such as the click stream, while others consider the scenario of weighted updates such as purchasing multiple units of the same item at once on an e-commerce platform. In this paper, we focus on the unit updates model and assume that items cannot be deleted if they were not previously inserted and hence all entries in frequency vector F are non-negative.

The frequency estimation problem takes an accuracy parameter ϵ and estimates the frequency of any item x such that $|\hat{f}(x) - f(x)| \leq \epsilon \cdot |F|_p$, where p can be either 1 or 2 corresponding to l_1 or l_2 norm and respectively provide l_1 or l_2 guarantees, $\hat{f}(x)$ is the estimated frequency and $f(x)$ is the actual frequency. When $p > 2$, providing l_p guarantee requires $poly(U)$ space [4]. In this paper, we focus on the l_1 problem variation. The ϕ frequent items problem is to identify a bag of *heavy items* whose frequency is greater than or equal to the specified threshold $\phi \cdot |F|_1$, where $0 < \phi < 1$. These heavy items are also known as the hot items.[3] In addition, some algorithms solve the (ϵ, ϕ)-approximate frequent items problem, which is to identify a bag of items B, given parameter $0 < \epsilon \leq \phi < 1$, such that B does not contain any element with frequency less then $(\phi - \epsilon)|F|_1$, i.e., $\forall i \in B$, $f(i) > (\phi - \epsilon)|F|_1$ and B contains all items with frequency greater than $\phi|F|_1$ i.e., $\forall i \notin B$, $f(i) < \phi|F|_1$.

2.2 Deterministic and Randomized Solutions

Reporting the exact frequent items requires $\Omega(N)$ space [13]. With limited memory, solving the exact frequent items problem is infeasible for large datasets. An alternative and practical approach in the context of big data is to use approximation techniques.

Deterministic solutions for the ϕ frequent items problem guarantee to return all heavy items and potentially some non-heavy items [21, 34, 38, 39]. Randomized solutions for the (ϵ, ϕ)-approximate frequent items problem allow the algorithm to fail with some probability δ [12, 17, 29]. In much of the literature, the failure probability is to set $\delta = O(U^{-c})$ where U is the bounded universe size and c is some constant. From the user perspective, deterministic algorithms provide stronger guarantees as all heavy items are identified. Randomized algorithms, on the other hand, with $1 - \delta$ probability report all heavy items and do not report any light weighted items.

2.3 Algorithms in Insertion-Only Model

The insertion-only model consists only of insert operations and many of the proposed algorithms in the insertion-only model are counter-based algorithms which maintain a fixed number of *item* and *count* pairs, and the underlying maintenance algorithm increments or decrements these counts to capture the frequency of items that are being tracked.

The first counter-based one pass algorithm to find the most frequent items in a large dataset dates back to the deterministic **Majority** Algorithm by Boyer and Moore in 1981 [9]. In 1982, Misra and Gries [39] generalized the majority problem and proposed the deterministic **MG** summary which uses $O(\frac{1}{\epsilon})$ space to solve the frequency estimation and frequent items problems. MG summary is a set of $\frac{1}{\epsilon}$ counters that correspond to monitored items. When a

[2] https://reports.collegeboard.org/pdf/2020-total-group-sat-suite-assessments-annual-report.pdf

[3] The term "Hot Items" was coined in [18]

Table 1: Comparison between different l_1 frequency estimation algorithm.

Sketch	Space	Update Time	Randomization	Model	Note
SpaceSaving [38]	$O(\frac{1}{\epsilon})$	$O(1)$	Deterministic	Insertion-Only	see Lemma 1
Count-Min [17]	$O(\frac{1}{\epsilon}log\frac{1}{\delta})$	$O(log\frac{1}{\delta})$	Randomized	Turnstile	Never Underestimate
Count-Median [12]	$O(\frac{1}{\epsilon}log\frac{1}{\delta})$	$O(log\frac{1}{\delta})$	Randomized	Turnstile	Unbiased Estimation
CSSampSim [29]	$O(\frac{1}{\epsilon}log\frac{1}{\delta}log\frac{\alpha logU}{\epsilon})$ bits	$\Theta(\frac{\alpha logU}{\epsilon U}log\frac{1}{\delta})$	Randomized	Bounded-Deletion	
Lazy-SpaceSaving$^{\pm}$	$O(\frac{\alpha}{\epsilon})$	$O(log\frac{\alpha}{\epsilon})$	Deterministic	Bounded-Deletion	see Lemma 7
SpaceSaving$^{\pm}$	$O(\frac{\alpha}{\epsilon})$	$O(log\frac{\alpha}{\epsilon})$	Deterministic	Bounded-Deletion	

Table 2: Frequently used symbols

σ	Data stream
N	Data stream length
$universe$	All data are drawn from $universe$
U	Size of the $universe$
F	Frequency vector
$f(x)$	x's true frequency
$\hat{f}(x)$	x's estimated frequency
ϵ	Accuracy
δ	Failure probability
I	Number of insertions
D	Number of deletions
α	In the bounded-deletion model, $D \leq (1-\frac{1}{\alpha})I$
$minCount$	The minimum count inside a sketch
$minItem$	The item with $minCount$

new item arrives, MG performs the following updates: if the new item is monitored, then increase its count by 1. Else if the summary is not full, monitor the new item. Else decrement all counts by 1 and remove any items with a count of zero. As a result of MG decrementing all counts by 1 when an arriving item is unmonitored, MG always underestimate item's frequency, and a hash-table implementation requires worst case $O(1/\epsilon)$ update time. Two decades later, Manku and Motwani [36] proposed a randomized **StickySampling** algorithm and a deterministic **LossyCounting** algorithm with worst case space $O(\frac{1}{\epsilon}log(\epsilon N))$, which exceeds the memory cost of MG summary. In 2003, Demaine et al. [21] and Karp et al. [34] independently generalized the majority algorithm and proposed the **Frequent** algorithm, which are both a rediscovery of the MG.

Two years later, in 2005, Metwally, Agrawal, and El Abbadi [38] proposed the **SpaceSaving** algorithm that provides highly accurate frequency estimates for frequent items and also presents a very efficient method to process insertions. SpaceSaving uses k counters to store an item's identity, estimated count and estimation error information, i.e., $(item, count_{item}, error_{item})$, and $error_{item}$ is an upper bound on the difference between the item's estimated frequency and its true frequency. When $k = \frac{1}{\epsilon}$, SpaceSaving solves both frequency estimation and frequent items problem. As shown in Algorithm 1, insertions proceed as follows, when a new item ($newItem$) arrives: if $newItem$ is monitored, then increment its count; if $newItem$ is not monitored and sketch size not full, then monitor $newItem$, and set $count_{newItem}$ to 1 and $error_{newItem}$ to 0; otherwise, SpaceSaving replaces the item ($minItem$) with the minimum count ($minCount$) by $newItem$, sets $error_{newItem}$ to $minCount$ and increments $count_{newItem}$. In SpaceSaving [38], $error_{item}$ is only used to show certain properties of the algorithm, while in this work we leverage this information for handling deletions. As shown in Algorithm 2, to estimate the frequency of an item in SpaceSaving, if the item is inside the sketch then report its count value, otherwise report 0. In [2], Agarwal et al. showed that both SpaceSaving and MG are mergeable [4], and a SpaceSaving with k counters can be isomorphically transformed into a MG summary with $k-1$ counters. Although SpaceSaving and MG share similarities, they follow different sets of update rules. When a new inserted item is unmonitored and the sketch is full, SpaceSaving replaces the min item with the new item and increments the count by one, whereas the MG decrements all item counts' by 1. As a result, SpaceSaving maintains an upper bound on the frequency of stored items, while MG always underestimates the frequency. Since SpaceSaving always increments one of the counts by one, the sum of all counts in SpaceSaving is equal to the $|F|_1$. Moreover, the SpaceSaving elegantly handles the case when an unmonitored new item arrives and the sketch is full, and naturally leads to a min-heap implementation such that incrementing any count and replacing the min item have $O(logk)$ update times, where k is the number of counters. SpaceSaving can also be implemented with a linked list data structure by keeping items with equal counts in a group, resulting in an $O(1)$ update time [38].

SpaceSaving satisfies the following properties (the first three properties are proved in [38] while the latter two are proved in [47]):

Lemma 1. *Frequency estimations for monitored items are never underestimated in SpaceSaving.*

Lemma 2. *SpaceSaving with $k = \frac{1}{\epsilon}$ counters ensures that after processing I insertions, the minimum count of all monitored items is no more than $\frac{I}{k} = \epsilon I$, i.e, $minCount < \epsilon I$.*

Lemma 3. *All items with frequency greater than or equal to minCount are inside the SpaceSaving.*

Lemma 4. *The sum of all estimation errors in the sketch is an upper bound on the sum of the frequencies of all unmonitored items.*

Lemma 5. *SpaceSaving with $\frac{1}{\epsilon}$ counters can estimate the frequency of any item with an additive error less than ϵI.*

Lemma 2 and Lemma 3, show that SpaveSaving with $\frac{1}{\epsilon}$ counters reports all items whose frequencies are greater than or equal to

[4]Mergeability is desired for distributed settings and means summaries over datasets can be merged into a single summary as if the single summary processed all datasets.

$\epsilon|F|_1$. Empirically, many studies have demonstrated that SpaceSaving outperforms other deterministic algorithms and it is considered to be the state of the art for finding frequent items [13, 35]. Moreover, due to the superior performance of SpaceSaving, many works use it as a fundamental building block [5, 42, 43, 45, 46]. Recently, a new randomized algorithm **BPtree** was proposed by Braverman et al. [11] to solve the frequent items problem with l_2 guarantees in the insertion-only model using $O(\frac{1}{\epsilon^2}log\frac{1}{\epsilon})$ space.

Algorithm 1: SpaceSaving Insert Algorithm

```
for item from insertions do
    if item ∈ Sketch then
        count_item += 1 ;
    else if Sketch not full then
        Sketch = Sketch ∪ item ;
        count_item = 1 ;
        error_item = 0 ;
    else
        // Sketch is full;
        minItem = min_{minItem∈Sketch} count_minItem;
        error_item = count_minItem ;
        count_item = count_minItem+1 ;
        Replace (minItem, count_minItem, error_minItem) by
          (item, count_item, error_item)
end
```

Algorithm 2: SpaceSaving Query(item)

```
if item ∈ Sketch then
    return count_item
return 0;
```

2.4 Algorithms in Turnstile Model

In the turnstile model, the stream consists of both insert and delete operations such that the deletes are always performed on previously inserted items. The sketches for solving the frequency estimation problem in the turnstile model are known as **linear sketches** [13]. While the counter-based solutions solve both the frequency estimation and frequent items problems, the linear sketch solutions directly answer the frequency estimation problem but need additional information to solve the frequent items problem. When assuming the inputs are from a bounded *universe*, linear sketches can query all items in the universe to identify the frequent items.

In 1999, Alon et al. [3] proposed the randomized **AMS** sketch to approximate the second frequency moment. Charikar et al. [12] improved upon the AMS sketch and proposed the randomized **Count-Median** sketch. The Count-Median sketch provides an unbiased estimator and uses $O(\frac{1}{\epsilon}log\frac{1}{\delta})$ and $O(\frac{1}{\epsilon^2}log\frac{1}{\delta})$ space to solve the l_1 and l_2 frequency estimation problems respectively. Cormode and Muthukrishnan [17] proposed the **Count-Min** sketch that shares a similar algorithm and data structure as the Count-Median sketch. Count-Min sketch never underestimates frequencies, and uses $O(\frac{1}{\epsilon}log\frac{1}{\delta})$ space to solve the l_1 frequency estimation problem.

Although one may exhaustively iterate through the universe to find frequent items, iterating through the universe can be slow and inefficient. As a result, Cormode and Muthukrishnan [17] suggested to imposes a hierarchical structure on the bounded universe, such that there are $logU$ layers and one Count-Min or Count-Median sketch per layer. Then use divide-and-conquer to search for the frequent items from the largest range to an individual item. The required space is $O(\frac{1}{\epsilon}log\frac{1}{\delta}logU)$ and update time is $O(log\frac{1}{\delta}logU)$. Dyadic intervals are in the form of $[i2^j, (i+1)2^j - 1]$ for $j \in log_2U$ and any constant i, such that any ranges can be decomposed into at most log_2U disjoint dyadic ranges [15]. Dyadic intervals over a bounded universe can be integrated with frequency estimation sketches to solve the quantile approximation problem in the turnstile model [17, 24, 44].

2.5 Algorithms in Bounded-Deletion Model

In the bounded-deletion model, the stream consists of both insert and delete operations and a constant $\alpha \geq 1$ is given such that at most $(1 - \frac{1}{\alpha})$ of prior insertions are deleted, i.e., $D \leq (1 - \frac{1}{\alpha})I$ where I is the number of insertions and D is the number of deletions. Jayaram et al. [29] proposed the CSSS (**C**ount-Median **S**ketch **S**ample **S**imulator) to solve the frequency estimation problem in the bounded-deletion model. Assuming $\delta = O(U^{-c})$ for some constant c and the maximum entry of F is $O(U)$, then the Count-Min and the Count-Median sketches require $O(\frac{1}{\epsilon}log^2U)$ bits and achieves the optimal space lower bound in the turnstile model [31]. Jayaram et al. [29] pointed out that in the bounded-deletion model by simulating the Count-Median sketch over $O(\frac{\alpha logU}{\epsilon})$ uniformly sampled items from the stream and then scaling the counts at the end, the *CSSS* sketch can accurately approximate the true frequency of an item with high probability. Hence, by carefully tuning the size of the Count-Median, CSSS requires $O(\frac{1}{\epsilon}log\frac{1}{\delta}log\frac{\alpha logU}{\epsilon})$ bits, improving the overall space compared to sketches in the turnstile model.

2.6 Summary

In Table 1, we compare the differences and similarities among several different sketches for l_1 frequency estimation. These sketches can also solve l_1 heavy hitters, though some sketches may need additional modifications to the parameters or leverage external data structures. In Table 2, we listed the important symbols used in the paper. Counter-based solutions have many advantages over linear sketches. Counter-based solutions are guaranteed to report all heavy items; they use $O(log\frac{1}{\epsilon})$ update time instead of $O(logU)$ update time where $\frac{1}{\epsilon}$ is often less than the universe size U; and they make no assumptions on the *universe* and thus can be useful in Big Data applications where items are drawn from unbounded domains. In this paper, we present SpaceSaving$^{\pm}$, an optimal counter-based deterministic algorithm with l_1 guarantee to solve both the frequency estimation and frequent items problem in the bounded-deletion model using $O(\frac{\alpha}{\epsilon})$ space.

3 THE SPACESAVING$^{\pm}$ ALGORITHM

In this section, we first show the space lower bound for solving the frequent items problem in the bounded-deletion model. Then, we introduce the Lazy *SpaceSaving*$^{\pm}$ and *SpaceSaving*$^{\pm}$ algorithms

with optimal space to solve both the frequency estimation and frequent items problems in the bounded-deletion model in which the total number of deletions (D) is less than $(1 - \frac{1}{\alpha})$ of the total insertions (I) where $\alpha \geq 1$. Given a user specified accuracy on the parameter ϵ, a deterministic algorithm for frequency estimation and frequent items problems must:

- Approximate the frequency of all items i with high accuracy such that $\forall\, i : |f(i) - \hat{f}(i)|) \leq \epsilon |F|_1$; and
- Report all the items with frequency greater than or equal to $\epsilon |F|_1$.

We propose Lazy SpaceSaving$^{\pm}$ and SpaceSaving $^{\pm}$. The main difference between the two variants of SpaceSaving$^{\pm}$ arises in the way deletions are handled. Since we assume the strict bounded-deletion model, a delete operation must correspond to a previously inserted item. If the item is being tracked in the sketch, processing such a delete operation is straightforward since the count associated with the item can be decreased by 1. On the other hand, the challenge arises when the sketch maintenance algorithm encounters a delete of an item that is not being tracked in the sketch. We develop different ways of handling such a delete in the two algorithms and the resulting correctness guarantees.

3.1 Space Lower Bound

We first show that there is no counter based algorithm that can solve the deterministic frequent items problem in the bounded-deletion model using less than $\frac{\alpha}{\epsilon}$ counters.

Theorem 1. *In the bounded-deletion model, any counter based algorithm needs at least $\frac{\alpha}{\epsilon}$ counters to solve the deterministic frequent items problem i.e, identify all the items with frequency greater than or equal to $\epsilon |F|_1$.*

Proof. By Contradiction.

Assume that there exists a counter based deterministic solution using $k < \frac{\alpha}{\epsilon}$ counters that can report all the items with frequency greater than or equal to $\epsilon |F|_1$. Consider a stream σ with bounded-deletions that contains I insertions and D deletions where all insertions come before any deletions. Let the I insertions consist exactly of $\frac{\alpha}{\epsilon}$ unique items, each with an exact count of $\frac{\epsilon}{\alpha} I$. After processing all insertions, the optimal algorithm with $k < \frac{\alpha}{\epsilon}$ counters will monitor at most k unique items, and there would be at least one item from the insertions that is left out. Let the set *Missing* contains all such unique items that appeared in I but are not monitored by the optimal algorithm. Now let the $D = (1 - \frac{1}{\alpha})I$ deletions be applied arbitrarily on the monitored items. After all D deletions, all items in *Missing* have frequency of $\frac{\epsilon}{\alpha} I$ in which $\frac{\epsilon}{\alpha} I \geq \epsilon(I - D) \geq \epsilon |F|_1$, and these items are frequent and must be monitored by the optimal algorithm. However, the sketch, with space k, after processing all insertions loses the information regarding *Missing*. Therefore, it is not possible to use less than $\frac{\alpha}{\epsilon}$ counters to solve the deterministic frequent items problem in the bounded-deletion model. □

3.2 Lazy SpaceSaving$^{\pm}$ Approach

Since supporting both insertions and bounded deletions is a much harder task compared to only allowing for insertions, the overall space needs to be increased. From the previous section, we can see that if the goal is to report all the items with frequency more than $\epsilon |F|_1$, where $|F|_1 = I - D$, we need to track more items. Since before any deletions, the sketch has no knowledge regarding which items are going to be deleted, then all elements with frequency higher than $\epsilon(I - D)$ are potential candidates before any deletions. We need an algorithm that can identify these potential candidate items.

By Lemma 2 and Lemma 3, SpaceSaving [38] with space k reports all the items with frequency greater than or equal to $\frac{I}{k}$. Therefore by using $k = \frac{\alpha}{\epsilon}$ space to process I insertions on the SpaceSaving algorithm, it will report all item with frequency greater than or equal to $\frac{\epsilon}{\alpha} I$. Since we know $\frac{1}{\alpha} \leq \frac{(I-D)}{I}$, $\frac{\epsilon}{\alpha} I \leq \epsilon I \frac{(I-D)}{I} = \epsilon(I - D)$. Hence by using $\frac{\alpha}{\epsilon}$ counters, all the items with frequency greater than or equal to $\epsilon(I - D)$ will be identified.

Interestingly, we find that modifying the original SpaceSaving algorithm with $O(\frac{\alpha}{\epsilon})$ space leads to an algorithm that solves the frequency estimation and frequent items problems in the bounded-deletion model. The Lazy SpaceSaving$^{\pm}$ algorithm handles insertions exactly in the same manner as in the original Algorithm 1. For deletions, the Lazy SpaceSaving$^{\pm}$ decreases the monitored item counter, if the deleted item is monitored. Otherwise, the deletions on unmonitored item are ignored, as shown in Algorithm 3. The frequency is still estimated according to Algorithm 2. The rationale for this design is that an unmonitored item has estimated frequency of 0 and deletions of the unmonitored items will not amplify the difference but in fact narrows the difference. Initially, this may seem to be counter-intuitive. Another way to think about it is that the frequency estimations for unmonitored items can only be underestimations. Thus, the decrease in an unmonitored item's exact frequency reduces the underestimation.

Algorithm 3: Lazy SpaceSaving$^{\pm}$: Deletion Handling

```
for item from deletions do
    if item in Sketch then
        count_item -= 1 ;
    else
        //ignore
end
```

We now formally establish that Algorithm 3 solves the frequency estimation problem in the bounded-deletion model. Let $error_{max}$ be the maximum error of frequency estimations in Lazy SpaceSaving$^{\pm}$. We show by induction that $error_{max}$ is always less than $\epsilon(I - D)$. First, we establish an upper bound on *minCount* in Lemma 6.

Lemma 6. *The minimum count, minCount, in Lazy SpaceSaving$^{\pm}$ with k counters is less than or equal to $\frac{I}{k}$.*

Proof. Since deletions never increment any counts, *minCount* is maximized by processing I insertions. With I insertions and no deletions, the sum of all counts is equal to I. *minCount* is largest when all the other counts are *minCount*. Hence, $minCount \leq \frac{I}{k}$. □

Theorem 2. *In the bounded-deletion model where $D \leq (1 - \frac{1}{\alpha})I$, after processing I insertions and D deletions, Lazy SpaceSaving$^{\pm}$ using $O(\frac{\alpha}{\epsilon})$ space solves the frequency estimation problem in which $\forall i, |f(i) - \hat{f}(i)| < \epsilon(I - D)$ where $f(i)$ and $\hat{f}(i)$ are the exact and estimated frequencies of an item i.*

Proof. By Induction.

Base case: After $i' < I$ insertions and 0 deletions with $O(\frac{\alpha}{\epsilon})$ space, we show that $error_{max}$ is less than $\epsilon(I-D)$ as follows: By Lemma 5 (of the insertion-only *SpaceSaving*), $error_{max} < i'\frac{\epsilon}{\alpha} \leq \epsilon\frac{i'(I-D)}{I} < \epsilon(I-D)$.

Induction hypothesis: After $i < I$ insertions and $d < D$ deletions, the maximum error of frequency estimations based on the sketch is $error_{max} < \epsilon(I-D)$.

Induction Step: Consider the case when the $(i+d+1)^{th}$ input item is an insertion. If the newly inserted item x is monitored or the sketch is not full, then no error is introduced. If the newly inserted item x is not monitored and the sketch is full, then x replaces the *minItem* which is the item with minimum count, *minCount*, in all monitored items. Based on Lemma 6, by using $\frac{\alpha}{\epsilon}$ counters in Lazy SpaceSaving$^{\pm}$, $minCount \leq i\frac{\epsilon}{\alpha} < \epsilon(I-D)$. The estimated frequency for x is *minCount*+1 and x is at most overestimated by *minCount*. The frequency estimation for *minItem* becomes 0, and *minItem*'s frequency estimation is off by at most *minCount*. Therefore, $error_{max}$ after processing the newly inserted item is still less than $\epsilon(I-D)$.

Consider the case when the $(i+d+1)^{th}$ input item is a deletion. If the newly deleted item x is monitored, its corresponding counter will be decremented and no extra error is introduced and $error_{max}$ is still less than $\epsilon(I-D)$. If the newly deleted item x is not monitored, then the algorithm ignores this deletion. The frequency estimation errors for monitored items do not change and they are still less than $\epsilon(I-D)$. Moreover, before the arrival of x, $\forall i \notin Sketch, f(i)-\hat{f}(i) = f(i)-0 < \epsilon(I-D)$. By ignoring the deletion of the unmonitored items, $\forall i \notin Sketch, (f(i)-1)-\hat{f}(i) < f(i)-\hat{f}(i) < \epsilon(I-D)$.

Conclusion: By the principle of induction, Lazy SpaceSaving$^{\pm}$ using $O(\frac{\alpha}{\epsilon})$ space solves the frequency estimation problem with bounded error, i.e, $\forall i, |f(i)-\hat{f}(i)| < \epsilon(I-D)$. □

Lazy SpaceSaving$^{\pm}$ also solves the frequent items problem. To prove this, we first show Lazy SpaceSaving$^{\pm}$ never underestimates the frequency of a monitored item.

Lemma 7. *Lazy SpaceSaving$^{\pm}$ never underestimates the frequency of monitored items.*

Proof. Since the handling of insertions is the same as the SpaceSaving and SpaceSaving never underestimates the frequency of monitored items by Lemma 1, it is clear that the insertions can not lead to frequency underestimation for monitored items. When handling deletions, Lazy SpaceSaving$^{\pm}$ only decrements the count when the deleted item is monitored. Since the deletion of a monitored item implies its exact frequency and its estimated frequency both decrease by one, this procedure has no effect on the difference between its exact frequency and estimated frequency. Therefore, Lazy SpaceSaving$^{\pm}$ never underestimates the frequency of monitored items. □

Since Lazy SpaceSaving$^{\pm}$ never underestimates, then report all the items with frequency estimations greater than or equal to $\epsilon(I-D)$, then all frequent items will be reported as shown in Theorem 3.

Theorem 3. *In the bounded-deletion model where $D \leq (1-\frac{1}{\alpha})I$, Lazy SpaceSaving$^{\pm}$ solves the frequent items problem using $O(\frac{\alpha}{\epsilon})$ space.*

Proof. By Contradiction.

Assume a frequent item x is not reported and by definition of frequent items, $f(x) \geq \epsilon(I-D)$. Since it is not reported, its frequency estimation, $\hat{f}(x)$, must be less than $\epsilon(I-D)$. There are two cases where x will not be reported: (i) x is not monitored, or (ii) x is monitored, but its frequency is underestimated, i.e., $\hat{f}(x) < \epsilon(I-D)$.

In the first case where x is not monitored, the estimation frequency of x is 0, i.e, $\hat{f}(x) = 0$. Since x is by assumption a frequent item, the frequency estimation difference for x is $|\hat{f}(x)-f(x)| \geq \epsilon(I-D)$. However, this contradicts Theorem 2 in which, for any items, the difference between its exact frequency and estimated frequency is less than $\epsilon(I-D)$.

In the second case, x is monitored but not reported which implies $\hat{f}(x)$ is less than $\epsilon(I-D)$. Since x is frequent, $\hat{f}(x)$ is an underestimation. However, by Lemma 7, Lazy SpaceSaving$^{\pm}$ never underestimates the frequency of monitored items.

Hence, by contradiction Lazy SpaceSaving$^{\pm}$ solves the deterministic frequent items problem. □

3.3 An illustration of Lazy SpaceSaving$^{\pm}$

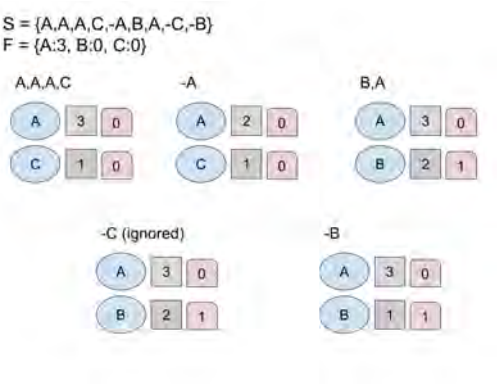


Figure 1: Input Stream consisting of 6 insertions and 3 deletions. Each tuple represents $(item, count_{item}, error_{item})$.

Consider an instance of Lazy SpaceSaving$^{\pm}$ with capacity of 2. The input stream σ is $(A, A, A, C, -A, B, A, -C, -B)$ where the minus sign indicate a deletion. The corresponding true frequency of A is 3 while the true frequency of all other items is 0. For the first four insertions and one deletion of the monitored item A, the sketch maintains the exact count with no errors. When the sixth item B arrives, B replaces item C, since C has the minimum count. The following insertion is A and since A is monitored, A's count increases. Then items $-C, -B$ arrive. Since C is not monitored, Lazy SpaceSaving$^{\pm}$ ignores the deletion of C, and the deletion of monitored item B decreases the corresponding count, as shown in Figure 1. After processing all inputs, the lazy-approach does not underestimate the frequency of the items in the sketch. It overestimates the frequency of item B in which $\hat{f}(B)-f(B) = 1$ and $\hat{f}(A)-f(A) = 0$.

3.4 SpaceSaving$^{\pm}$

While Lazy SpaceSaving$^{\pm}$ elegantly satisfies all the necessary requirements, the average frequency error and total frequency error

may increase if there are significant deletions of the unmonitored items. Therefore, we propose SpaceSaving$^{\pm}$, a novel algorithm and data structures that accurately handles deletions of the unmonitored items and item's frequency is still estimated according to Algorithm 2. Interestingly, we experimentally show that SpaceSaving$^{\pm}$ performs better than Lazy SpaceSaving$^{\pm}$ when they are both allocated the same sketch space, even though we need more space by a constant factor to establish the correctness of SpaceSaving$^{\pm}$.

Both SpaceSaving and our proposed Lazy SpaceSaving$^{\pm}$ algorithms have the property of never underestimating the frequency of the monitored items. Since the ϵ-approximation requirement is $\forall i, |f(i) - \hat{f}(i)| < \epsilon(I - D)$, there are opportunities to reduce the amount of overestimation for the monitored items, as long as the difference is still within this bound. We observe that an item with a large estimation error indicates that it is unlikely to be a heavy item, as heavy items are often never evicted from the sketch and have small estimation error. In addition, items with large estimation errors are often overestimated due to the aggregation of the frequencies of many less-weighted items. SpaceSaving$^{\pm}$ leverages this intuition. It handles the insertions of all items, and the deletions of the monitored items exactly in the same way as the Lazy SpaceSaving$^{\pm}$. For the deletions of the unmonitored items, SpaceSaving$^{\pm}$ decrements the count of the item that has the maximum estimation error inside the sketch, as shown in Algorithm 4. The estimation error in SpaceSaving$^{\pm}$ is an upper bound on the difference between the item's estimated frequency and its true frequency. With this modification, the estimated frequency of any item reduces either from being replaced or from a deletion of an unmonitored item. In the following proofs, SpaceSaving$^{\pm}$ uses $\frac{2\alpha}{\epsilon}$ to ensure (i) no item can be severely overestimated, and (ii) no item can be severely underestimated. To estimate the frequency of an item, we still use Algorithm 2. Before analyzing the correctness of the algorithm, we first establish three helpful lemmas.

Lemma 8. *The minimum count, minCount, in SpaceSaving$^{\pm}$ with $\frac{2\alpha}{\epsilon}$ counters is less than or equal to $\frac{\epsilon}{2}(I - D)$.*

Proof. Similar to the proof of Lemma 6. Since deletions never increment any counts, *minCount* is maximized by processing I insertions and hence the sum of all the counts is upper bounded by I. *minCount* is largest when all the other counts are also *minCount*. Hence, $minCount \leq \frac{\epsilon I}{2\alpha} \leq \frac{\epsilon(I-D)}{2}$. □

Lemma 9. *The maximum estimation error, $argmax_{j \in Sketch} error_j$, in SpaceSaving$^{\pm}$ with $\frac{2\alpha}{\epsilon}$ counters is less than $\frac{\epsilon}{2}(I - D)$.*

Proof. The estimation error only increases when *minItem* is replaced by a newly inserted item and after the replacement, the estimation error becomes *minCount*. *minCount* is maximized when the input contains I insertions and no deletions. Hence by Lemma 2, SpaceSaving$^{\pm}$ with $\frac{2\alpha}{\epsilon}$ counters has $minCount < \frac{\epsilon}{2}(I - D)$. The estimation error is at most *minCount* and less than $\frac{\epsilon}{2}(I - D)$. □

Lemma 10. *The sum of all estimation errors in SpaceSaving$^{\pm}$, is an upper bound on the sum of frequencies of all unmonitored items and the maximum estimation error is greater than or equal to 0.*

Proof. The deletion of a monitored item has no effect on the sum of the estimation errors, and it has no effect on the sum of the frequencies of the unmonitored items. The deletion of an unmonitored item decreases both the sum of the frequencies of the unmonitored items by 1 and the sum of the estimation errors by 1. From this observation and Lemma 4, we can conclude that in SpaceSaving$^{\pm}$ with k counters, the sum of all estimation errors is an upper bound on the sum of frequencies of all unmonitored items. Since the sum of frequencies of all unmonitored items is greater than or equal to 0 and the sum of all estimation errors is upper bounded by k times the maximum estimation error, the maximum estimation error is greater than or equal to 0. □

Algorithm 4: SpaceSaving$^{\pm}$: Deletion Handling

```
1 for item from deletions do
2     if item in Sketch then
3         count_item -= 1 ;
4     else
5         j = arg max_{j∈Sketch} error_j ;
6         count_j -= 1 ;
7         error_j -= 1 ;
8 end
```

Theorem 4. *In the bounded-deletion model where $D \leq (1 - 1/\alpha)I$, after processing I insertions and D deletions, SpaceSaving$^{\pm}$ using $O(\frac{\alpha}{\epsilon})$ space solves the frequency estimation problem in which $\forall i, |f(i) - \hat{f}(i)| < \epsilon(I - D)$ where $f(i)$ and $\hat{f}(i)$ are the exact and estimated frequencies of an item i.*

Proof. Consider an instance of SpaceSaving$^{\pm}$ with $\frac{2\alpha}{\epsilon}$ counters to process I insertions and D deletions. First, we prove there is no item i such that the frequency estimate of i severely overestimate its true frequency, i.e, $\nexists i, \hat{f}(i) - f(i) > \epsilon(I - D)$. In SpaceSaving$^{\pm}$, the handling of deletions can not lead to any overestimation as counters will only be decremented, and only the replacement of the *minItem* due to a newly inserted item can lead to frequency overestimation of the newly inserted item. From Lemma 8, the *minCount* in SpaceSaving$^{\pm}$ with $\frac{2\alpha}{\epsilon}$ counters is no more than $\frac{\epsilon}{2}(I - D)$. The overestimation of a newly inserted item can be at most *minCount*. Therefore, no item can be overestimated by more than $\frac{\epsilon}{2}(I - D)$.

Second, we prove there is no item that can be severely underestimated i.e, $\nexists i, \hat{f}(i) - f(i) < -\epsilon(I - D)$. Two operations may lead to frequency underestimation: (i) Replacement of *minItem*, or (ii) Deletion of an unmonitored item. For the first case, *minCount* is always less than $\frac{\epsilon}{2}(I - D)$, and the amount of underestimation is less than $\frac{\epsilon}{2}(I - D)$ for any item due to the replacement.

We show that the deletion of an unmonitored item can lead to at most $\frac{\epsilon}{2}(I - D)$ frequency underestimation. Based on Lemma 9 and Lemma 10, the maximum estimation error must be less than $\frac{\epsilon}{2}(I - D)$ and greater than or equal to 0. In Algorithm 4, lines 6 and 7, the deletion of an unmonitored item decreases both the count and the estimation error of the item with the maximum estimation error. Call this item x. Since x's counter decreases by 1, the difference between x's frequency estimation and x's true frequency, $\hat{f}(x) - f(x)$, also decreases by 1. Once an item becomes

monitored, its estimation error can only decrease. The number of decrements due to an unmonitored item is at most $\frac{\epsilon}{2}(I-D)$. Hence for any item, its frequency is underestimated by at most $\frac{\epsilon}{2}(I-D)$ due to the deletion of unmonitored items. As a result, for any item, its frequency can be underestimated by at most $\epsilon(I-D)$ by replacing the $minItem$ and the deletions of the unmonitored items. □

In Theorem 4, we proved that SpaceSaving$^{\pm}$ guarantees that all items' estimated frequencies are off by no more than $\epsilon(I-D)$, i.e., $\forall i, |\hat{f}(i)-f(i)| \leq \epsilon(I-D)$. By reporting all the items with estimated frequency greater than 0, all frequent items must be reported, which is established in Theorem 5.

Theorem 5. *In the bounded-deletion model, where $D \leq (1-\frac{1}{\alpha})I$, SpaceSaving$^{\pm}$ solves the frequent items problem using $O(\frac{\alpha}{\epsilon})$ space.*

Proof. Proof by contradiction:

Assume SpaceSaving$^{\pm}$ does not report all frequent items. There must exist a frequent item x that is not reported. Since SpaceSaving$^{\pm}$ reports all the items with estimated frequency greater than 0 as frequent items (recall unmonitored items have estimated frequency of 0), x's estimated frequency must be less than or equal to 0, i.e., $\hat{f}(x) \leq 0$. Moreover, since x is a frequent item, then the exact frequency of x must be greater than or equal to $\epsilon(I-D)$, i.e., $f(x) \geq \epsilon(I-D)$. The difference between x estimated frequency and x exact frequency is greater than or equal to $\epsilon(I-D)$, i.e., $|f(x)-\hat{f}(x)| \geq \epsilon(I-D)$. This leads to a contradiction since it violates the frequency approximation guarantee proved in Theorem 4. □

3.5 An illustration of SpaceSaving$^{\pm}$

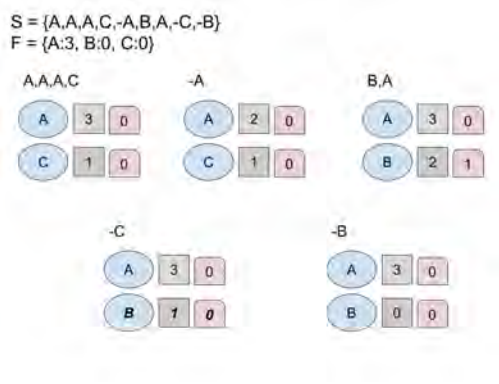


Figure 2: Input Stream consists of 6 insertions and 3 deletions. Each tuple represents $(item, count_{item}, error_{item})$.

Consider the same stream illustrated in Section 3.3 in which the stream σ is $(A, A, A, C, -A, B, A, -C, -B)$ where the minus sign indicate a deletion. The corresponding exact frequency of A is 3, while the true frequency of all other items is 0. Consider an instance of SpaceSaving$^{\pm}$ with capacity of 2. The SpaceSaving$^{\pm}$ image after digesting the first 7 items are exactly the same as in the previous example. When the deletion of item C arrives, SpaceSaving$^{\pm}$ does not ignore the deletion of unmonitored item C, and since item B has the largest estimation error, both B's count and B's estimation error are decreased. The final deletion of A decreased A's corresponding count. After processing the stream, the estimated frequency for A and B are 3 and 0 respectively, as shown in Figure 2. The frequency estimations are exact in which $|\hat{f}(A)-f(A)| = 0$ and $|\hat{f}(B)-f(B)| = 0$. With the same bounded-deletion stream and sketch space, Lazy SpaceSaving$^{\pm}$ overestimated the frequency of item B by 1 (Section 3.3), while SpaceSaving$^{\pm}$ is able to further reduce the estimation error to 0. By judiciously handling the deletion of the unmonitored items, SpaceSaving$^{\pm}$ reduces the impact of overestimation and achieves better accuracy.

3.6 Min Heap and Max Heap

SpaceSaving algorithm is usually implemented with a standard min-heap data structure such that the operations that increase the item counts and that remove the minimum item can be performed in logarithmic time [6]. To support the deletion of the unmonitored items, SpaceSaving$^{\pm}$ further needs to find the item with the maximum estimation error and modify the estimation errors efficiently. From these observations, we use two heaps on both the estimated counts and the estimation errors, as underlying data structures. The estimated counts are stored in a min heap, the estimation errors are stored in a max heap, and a dictionary maps each item to the corresponding nodes in these two heaps. Using two heaps and a dictionary with $O(k)$ space, both the minimum count and maximum estimation error can be found in $O(1)$ time; while insertions and deletions can be done in $O(logk)$ time. For example, if the sketch needs to delete an unmonitored item, then the procedure would be as follows: (1) use the dictionary to ascertain that the deletion is performed on an unmonitored item; (2) use the max heap to find the item with maximum estimation error; (3) use the dictionary to find the location of the item with maximum estimation error in the min heap; (4) decrease both its count (min heap) and its estimation error (max heap); (5) percolate it up in min heap and percolate it down in max heap;

4 EVALUATION

This section evaluates the performance of Lazy SpaceSaving$^{\pm}$ and SpaceSaving$^{\pm}$. They are the first deterministic frequency estimation and frequent item algorithms in the bounded-deletion model and make no assumptions on the universe. The experiments aim to identify advantages and disadvantages of Lazy SpaceSaving$^{\pm}$ and SpaceSaving$^{\pm}$ compared to other state-of-the-art sketches such as:

- **CSSS** [29] : The CSSS sketch is the first theoretical algorithm to solve the frequency estimation and frequent item problems in the bounded-deletion model.
- **Count-Min** [5] [17]: Count-Min Sketch operates in the turnstile model and never underestimate frequencies.
- **Count-Median** [6] [12]: Count-Median Sketch operates in the turnstile model and its frequency estimation is unbiased.

4.1 Experimental Setup

We implemented SpaceSaving$^{\pm}$ using the min and max heap data structures described in Section 3.6 in Python. The main distinction from the original SpaceSaving [38] are: (i) support of delete

[5] See https://github.com/rafacarrascosa/countminsketch for implementation detail
[6] See [19] for implementation detail

operations using the Algorithm 3 or Algorithm 4; (ii) the use of a max heap on the estimation errors; and (iii) the overall space complexity is $\frac{\alpha}{\epsilon}$ and the update time complexity is $O(log\frac{\alpha}{\epsilon})$. We also implemented the CSSS sketch as described in [29]. All the experimental metrics are averaged over 5 independent runs. In all experiments, Lazy SpaceSaving$^{\pm}$ and SpaceSaving$^{\pm}$ use the same amount of space, and to align the experiments with the theoretical literature [7, 29], we set the universe size $U = 2^{16}$ and $\delta = U^{-1}$.

4.2 Data Sets

The experimental evaluation is conducted using both synthetic and real world data sets consisting of items that are inserted and deleted. For the synthetic data, we consider three different distributions:

- **Zipf Distribution**: The elements are drawn from a bounded universe and the frequencies of elements follow the Zipf Law [49], in which the frequency of an element with rank R: $f(R, s) = \frac{constant}{R^s}$ where s indicates skewness. Deletions are uniformly chosen from the insertions.
- **Binomial Distribution**: The elements are generated according to the binomial distribution with parameters n and p where p is the probability of success in n independent Bernoulli trials.

In addition to the synthetic data sets, we used the following real world CAIDA Anonymized Internet Trace 2015 Dataset [1].

- **2015 CAIDA Dataset**: The CAIDA dataset is collected from the 'equinixchicago' high-speed monitor. In the experiment, we use 5 disjoint batches of 2 million TCP packets where insertions are the destination IP addresses and deletions are randomly chosen from insertions.

We also conducted experiments by exploring two additional patterns of the data sets:

- **Shuffled**: The insertions are randomly shuffled and the deletions are randomly and uniformly chosen from insertions.
- **Targeted**: The insertions are randomly shuffled and the deletions delete the item with the least frequency.

The metrics used in the experiments are:

- **Mean Squared Error**: The mean squared error (MSE) is the average of the squares of the frequency estimation errors. MSE is a measurement widely used to judge the accuracy of an estimation and serves as an empirical estimation of the variance [16].
- **Recall**: Recall is defined as $\frac{TP}{TP+FN}$ where TP (true positive) is the number of items that are estimated to be frequent and are indeed frequent and FN (false negative) is the number of items that are frequent but not included in the estimations.
- **Precision**: Precision is defined as $\frac{TP}{TP+FP}$ where FP (false positive) is the number of items that are estimated to be frequent but are not frequent.

The experiments are presented in the following two subsections: frequency estimation and frequent item experiments.

4.3 Frequency Estimation Evaluation

In this section, we compare Lazy SpaceSaving$^{\pm}$ and SpaceSaving$^{\pm}$ with state-of-the-art frequency estimation sketches. Our proposed algorithms use $O(\frac{\alpha}{\epsilon})$ space while the Count-Min and Count-Median use $O(\frac{1}{\epsilon}logU)$ space. When $\alpha = logU$, they share the same space and the delete:insert ratio becomes $\frac{logU-1}{logU}$. In addition, these experiments evaluate the accuracy of each sketch using the mean square error (MSE). In MSE figures the x-axis denotes the sketch size while the y-axis depicts the average of the mean square errors. Since the mean square error is strictly positive, the lower y-axis values indicate better accuracy. In the following experiments, we assume all insertions arrive before any deletions into the sketch which is an adversarial pattern as spatial locality is minimized.

4.3.1 Sketch Size. In this experiment, the input data has I insertions and D deletions. The delete:insert ratio is 0.5 and *alpha* equals to 2. The deletion pattern is either shuffled, randomly chosen from insertions, or targeted delete of the least frequent items. The Zipf and Binomial distributions have $|F|_1 = 10^5$ and the CAIDA dataset has $|F|_1 = 10^6$. This experiment explores the effect of distribution skewness and the space size effect of sketches operating in both the bounded-deletion model and in the turnstile model.

As expected, all sketches share the same pattern: increasing the sketch size leads to decrease in the MSE, shown in Figure 3. All experiments show SpaceSaving$^{\pm}$ has the lowest MSE and best accuracy as the sketch size grows. For the skewed Zipf distribution and CAIDA dataset, SpaceSaving$^{\pm}$ is the clear winner for all sketch sizes, as shown in Figure 3. For the lesser skewed binomial distribution, Count-Median performs competitively compared to SpaceSaving$^{\pm}$; however, SpaceSaving$^{\pm}$ eventually has better accuracy as the sketch size increases, as shown in Figure 3(b,e). The CSSS sketch has accuracy between Count-Median and Count-Min sketches. The Count-Min sketch often overestimates an item's frequency and thus has higher mean square error across all distribution.

The targeted deletion pattern, when the least frequent items are targeted for deletions, leads to a slight decrease in MSE across all distributions for Count-Min. The targeted delete pattern decreases the cardinality of F, increases the overall skewness, and hence heavy hitter items become more dominant and all sketches are able to capture the overall change and have less mean square error.

4.3.2 Delete:Insert Ratio and α. Sketches in the bounded-deletion model have space complexity dependent on parameter α, which upper bounds the delete:insert ratio. With higher delete:insert ratio, these sketches need to increase their sketch space to tolerate the increase in deletions in order to deliver the same guarantee. In this experiment, we fixed the sketch space to $10^3 logU$ bits and fixed the input stream length to one million. The x-axis represents different delete:insert ratio, and the y-axis is the mean squared error averaged over 5 independent runs, as shown in Figure 4.

As expected, the accuracy of Lazy SpaceSaving$^{\pm}$ and CSSS depends on α and their MSE increases as the delete:insert ratio increases. The more interesting result is that SpaceSaving$^{\pm}$'s MSE decreases when the delete:insert ratio is less or equal to 0.9. Moreover, for a universe of size 2^{16}, SpaceSaving$^{\pm}$ provides MSE less than CSSS, Count-Min, and Count-Median even if the delete:insert ratio is as high as 0.9375, which is $\frac{logU-1}{logU}$, while using the same amount of space, as shown by the right most values in Figure 4. By handling the deletion of unmonitored items judiciously, SpaceSaving$^{\pm}$'s frequency estimation is more robust to the increase in deletions than other algorithms in the bounded-deletion model. For sketches

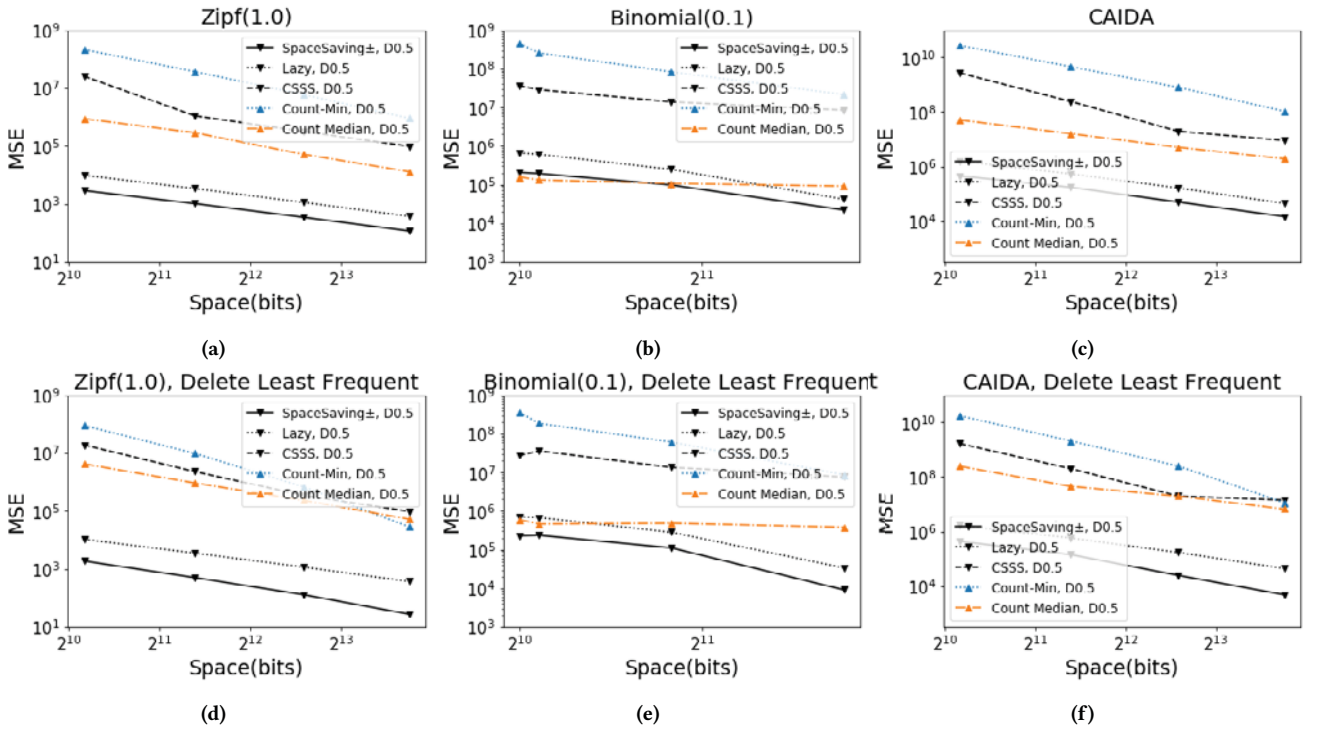


Figure 3: Trade-off between space and accuracy on various data distributions and different patterns

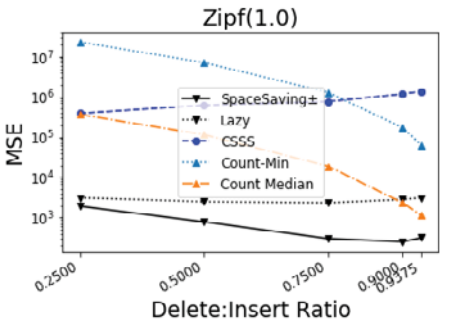


Figure 4: Varying delete:insert ratio.

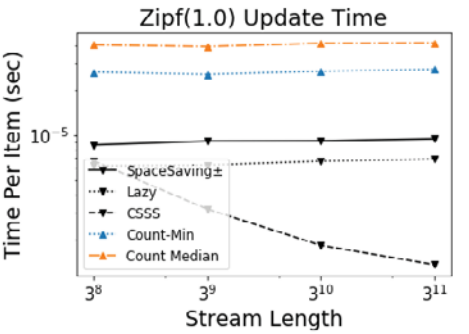


Figure 5: Update times for Sketches

that operate in the turnstile model, the MSE of Count-Min and Count-Median decreases as the delete:insert ratio increases, since more deletions reduce the number of hash collisions and reduce the amount of over counting in each bucket. If the universe size increases, the performance of linear sketches will further decrease, whereas the data-driven SpaceSaving$^{\pm}$ has no dependency on the universe, and can provide accurate estimations even in the extreme case of unbounded universe.

4.3.3 Update Time. In Figure 5 , the x-axis is the stream length and the y-axis is the average latency in second per item over 5 independent runs. The input is a shuffled Zipf distribution and the delete:insert ratio is 0.5. All sketches use $10^3 logU$ bits. As shown in Figure 5, Lazy SpaceSaving$^{\pm}$ has slightly less update time than SpaceSaving$^{\pm}$. The lazy approach ignores deletions of unmonitored items and achieves better latency. CSSS sketch update time decreases as the stream length grows because it performs sampling to obtain $O(\frac{\alpha logU}{\epsilon})$ samples and runs Count-Median sketch on the samples. As the stream length increases the sample size increases at a slower pace, and the average update time per item decreases. Count-Min and Count-Median have update times depend on the universe size where a larger universe size will further increase the update time. Since Count-Median performs more hashes than Count-Min, Count-Median requires more update time than Count-Min. While randomized CSSS has fast update time, our algorithms are deterministic and provide very accurate approximations.

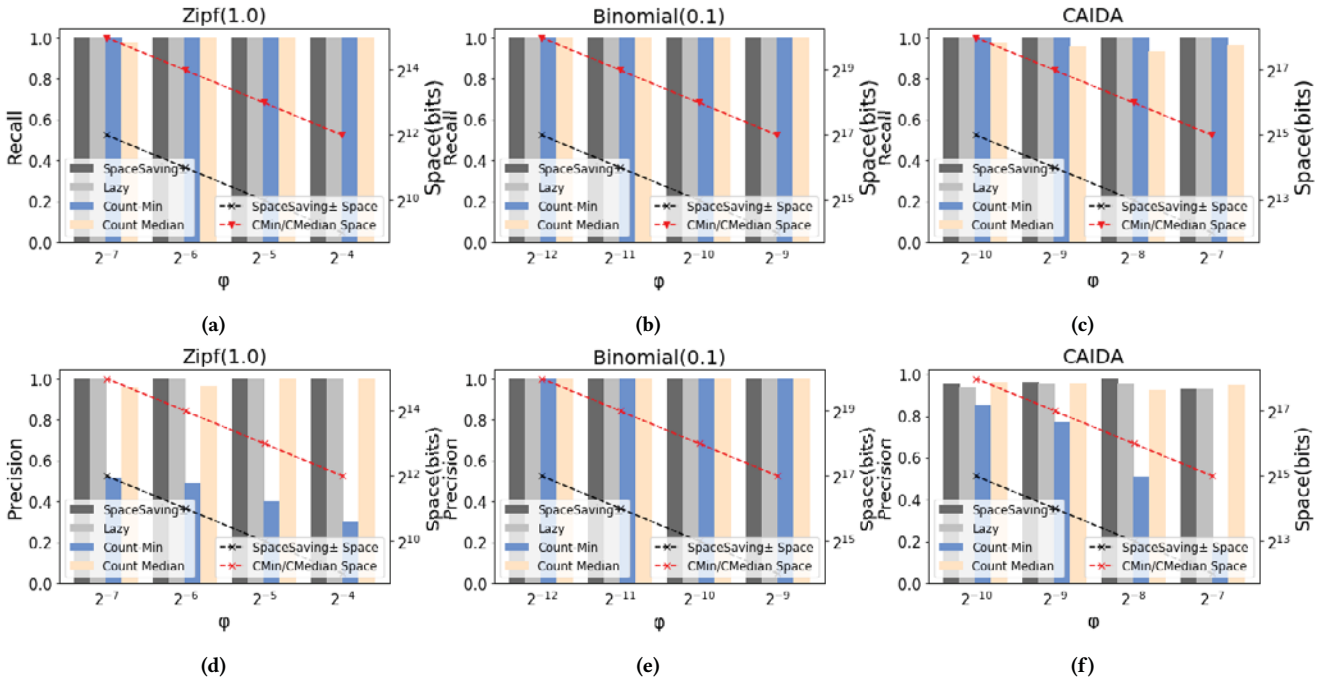


Figure 6: Recall and Precision Comparison

4.4 Frequent Items Evaluation

In this section, we compare the recall and precision of our proposed algorithms with state-of-the-art sketches for identifying frequent items. All experiments in this section have delete:insert ratio of 0.5, $\alpha = 2$, and all insertions arrive before any deletions. The left y-axis depicts either the average recall or average precision over 5 independent runs: higher y-axis values indicate better recall or precision. The right y-axis denotes the space used for each sketch where Lazy SpaceSaving$^{\pm}$ and SpaceSaving$^{\pm}$ use $\frac{\alpha}{\epsilon} logU$ bits; Count-Min and Count-Median use $\frac{1}{\epsilon} log^2 U$ bits. Each sketch queries all potential items and then reports items with estimated frequency greater than or equal to $\phi|F|_1$ as the frequent items. In addition, the following experiments do not compare with the CSSS sketch. Although CSSS can solve the frequent item problem, CSSS is more of theoretical interest since it reduces the size of each counter from $O(logU)$ bits to $O(log(\alpha))$ bits but in practice, it requires a lot more space to solve the frequent item problem. More specifically, the sketch size increases by 192 times, which implies the universe is powered by 192 times. The space increase is more significant than the space saved by reducing the number of bits per counter.

4.4.1 Recall. In these experiments, we compare the recall among Lazy SpaceSaveing$^{\pm}$, SpaceSaving$^{\pm}$, Count-Min and Count-Median. In Figure 6 (a), (b), and (c), the x-axis represents different frequent items threshold ϕ in which frequent items have frequency greater than or equal to $\phi|F|_1$. The right y-axis denotes the space used for each sketches in which Lazy SpaceSaving$^{\pm}$ and SpaceSaving$^{\pm}$ use $\frac{\alpha}{\epsilon} logU$ bits; Count-Min and Count-Median use $\frac{1}{\epsilon} log^2 U$ bits. The sketch space increases as ϕ decreases. The left y-axis is the recall ratio. As expected, Lazy SpaceSaving$^{\pm}$ and Count-Min sketches have 100% recall across all distributions, since they never underestimate the frequent item's frequency. The Count-Median sketch may sometimes underestimate the frequency and thus does not always achieve 100% recall, as shown in Figure 6 (c). In the proof of Theorem 5, SpaceSaving$^{\pm}$ needs to report all items with frequency greater than 0 to identify all frequent items and achieve 100% recall. In this experiment, SpaceSaving$^{\pm}$ reports items with frequency larger than $\phi|F|_1$. Since it might underestimate an item's frequency, the recall rate might not be 100%. However, in these experiments, SpaceSaving$^{\pm}$ still achieves 100% recall across all distributions and thus indicates SpaceSaving$^{\pm}$ rarely underestimates.

4.4.2 Precision. In this subsection, we compare the precision among Lazy SpaceSaveing$^{\pm}$, SpaceSaving$^{\pm}$, Count-Min and Count-Median. In Figure 6 (d), (e), and (f), the x-axis represents the different frequent items threshold ϕ. The right y-axis denotes the space budget in which Lazy SpaceSaving$^{\pm}$ and SpaceSaving$^{\pm}$ use $\frac{\alpha}{\epsilon} logU$ bits; Count-Min and Count-Median use $\frac{1}{\epsilon} log^2 U$ bits. The sketch space increases as ϕ decreases. The left y-axis is the precision ratio. Lazy SpaceSaving$^{\pm}$, SpaceSaving$^{\pm}$ and Count-Median have above 90% precision for all ϕ and distributions. Since Lazy SpaceSaving$^{\pm}$ sometimes overestimates an item's frequency, a few items' frequency are overestimated and hence they may be falsely classified as frequent items. SpaceSaving$^{\pm}$ judiciously handles the deletion and achieves very high precision across all distributions using minimal space. Count-Min often overestimates items' frequencies and many items are incorrectly classified as frequent.

5 QUANTILE SKETCH

In this section, we demonstrate that SpaceSaving$^{\pm}$ can be easily integrated with prior protocols [17, 44] to solve the quantile approximation problem. We propose Dyadic SpaceSaving$^{\pm}$ (DSS$^{\pm}$), the first deterministic quantile sketch in the bounded-deletion model. Dyadic SpaceSaving$^{\pm}$ sketch is a universe-driven algorithm that accurately approximates quantiles with strong guarantees.

5.1 The Quantiles Problem

The rank of an element x is the total number of elements that are less than or equal to x, denoted as $R(x)$. The quantile of an element x is defined as $R(x)/|F|_1$ where F is the frequency vector. The most familiar quantile value is 0.5 also known as *median. Deterministic* ϵ approximation quantile algorithms [25, 41] take as input a precision value ϵ and an item such that the approximated rank has at most $\epsilon|F|_1$ additive error. The *randomized* quantile algorithms provide a weaker guarantee in which the approximated rank of an item has at most $\epsilon|F|_1$ additive error with high probability [28, 33, 37].

Recently, Zhao et al. [48] proposed the first randomized quantile sketch KLL$^{\pm}$ using $O(\frac{\alpha^{1.5}}{\epsilon}\ log^2 log\frac{1}{\epsilon})$ space in the bounded deletion model by generalizing the KLL [33] from the insertion-only model. The first sketch to summarize quantiles in the turnstile model is the Random Subset Sums (RSS) proposed by Gilbert et al. [24]. RSS is a universe driven algorithm, which assume input are drawn from a bounded universe and maintain attributes over the bounded universe [14]. RSS breaks down the bounded universe into dyadic intervals and maintains frequency estimations for each interval. Recall, dyadic intervals are in the form of $[i2^j, (i+1)2^j - 1]$ for $j \in log_2 U$ and any constant i, such that any ranges can be decomposed into at most $log_2 U$ disjoint dyadic ranges [15]. Cormode et al. [17] proposed the Dyadic Count-Min (DCM) which replaces the frequency estimation sketch for each dyadic interval with a Count-Min, and hence improves the overall space complexity to $O(\frac{1}{\epsilon}\log^2 U \log(\frac{\log U}{\epsilon})))$ and update time to $O(\log U \log(\frac{\log U}{\epsilon}))$. Then, Wang et al. [44] proposed the Dyadic Count-Median (DCS) which replaces Count-Min with Count-Median [12] to further improve the space complexity to $O(\frac{1}{\epsilon}\log^{1.5} U \log^{1.5}(\frac{\log U}{\epsilon})))$, while using the same update time complexity as DCM.

5.2 DSS$^{\pm}$: A Deterministic Quantile Sketch

We propose the Dyadic SpaceSaving$^{\pm}$ (DSS$^{\pm}$) to solve deterministic quantile approximation in the bounded-deletion model. Inspired by the previous algorithms, we observe that by replacing the frequency estimation sketch in each dyadic layer with a SpaceSaving$^{\pm}$ of space $O(\frac{\alpha}{\epsilon} logU)$ solves the quantile approximation in the bounded-deletion model. Any range can be decomposed into at most $logU$ dyadic intervals [15]. Since SpaceSaving$^{\pm}$ with $O(\frac{\alpha}{\epsilon} logU)$ space ensures that the frequency estimation has at most $\frac{\epsilon(I-D)}{logU}$ additive error and by summing up at most $logU$ frequencies, the approximated rank has at most $\epsilon(I-D)$ additive error and the approximated quantile has at most ϵ error. To update the DSS$^{\pm}$ quantile sketch with an item x: for each $logU$ layers, x is mapped to an element in that layer and increments the corresponding element's frequency, as shown in Algorithm 5. The rank information of an item can be calculated by summing $O(logU)$ number of subset sums, as shown in Algorithm 6. Therefore, the Dyadic SpaceSaving$^{\pm}$ sketch requires $O(\frac{\alpha}{\epsilon} log^2 U)$ space with update time $O(logU log\frac{\alpha logU}{\epsilon})$. The quantile experiments comparing DSS$^{\pm}$, KLL$^{\pm}$ and DCS are shown in [47].

Algorithm 5: DSS$^{\pm}$ Update(x,1)

```
for h from 0 to logU do
    DSS±[h].update(x, 1);
    x= x/2;
end
```

Algorithm 6: DSS$^{\pm}$ Query(x)

```
Rank = 0;
for h from 0 to logU do
    if x is odd then
        Rank = Rank + DSS±[h].query(x);
    x= x/2;
end
return Rank;
```

6 CONCLUSION

Frequency estimation and frequent items are two important problems in data stream research, and have significant impact for real world systems. Over the past decades of research, many algorithms have been proposed for the insertion-only and the turnstile models. In this work, we propose data-driven deterministic SpaceSaving$^{\pm}$ sketches to accurately approximate item frequency and report heavy hitter items in the bounded-deletion model. To our knowledge, Lazy SpaceSaving$^{\pm}$ and SpaceSaving$^{\pm}$ are the first deterministic algorithms to solve these two problems in the bounded-deletion model and they make no assumption on the universe. The experimental evaluations of SpaceSaving$^{\pm}$ highlight that it has the best frequency estimation accuracy among other state-of-the-art sketches, and requires the least space to provide strong guarantees. We also demonstrate that implementing SpaceSaving$^{\pm}$ with the min and max heap approach provides fast update time. Furthermore, the experiments showcase that SpaceSaving$^{\pm}$ has very high recall and precision rates across a range of data distributions. These characteristics of SpaceSaving$^{\pm}$ make it a practical choice for real world applications. Finally, by leveraging SpaceSaving$^{\pm}$ and dyadic intervals over bounded universe, we proposed the first deterministic quantile sketch in the bounded-deletion model. Our analysis clearly demonstrates that overall, for an unbounded universe or for practical delete:insert ratios below $\frac{logU-1}{logU}$ (e.g., for a realistic universe size of U=2^{16}, a ratio of .93 and for U=2^{32}, a ratio of .96), SpaceSaving$^{\pm}$ is the best algorithm to use and solves several major problems with strong guarantees in a unified algorithm.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback and also thank Rajesh Jayaram and Dan Qiao for helpful discussions. This work is funded in part by NSF grants CNS-1703560 and CNS-1815733.

REFERENCES

[1] [n.d.]. Anonymized Internet Traces 2015. https://catalog.caida.org/details/dataset/passive_2015_pcap. Accessed: 2021-11-5.

[2] Pankaj K Agarwal, Graham Cormode, Zengfeng Huang, Jeff Phillips, Zhewei Wei, and Ke Yi. 2012. Mergeable summaries. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI symposium on Principles of Database Systems*. 23–34.

[3] Noga Alon, Yossi Matias, and Mario Szegedy. 1999. The space complexity of approximating the frequency moments. *Journal of Computer and system sciences* 58, 1 (1999), 137–147.

[4] Ziv Bar-Yossef, Thathachar S Jayram, Ravi Kumar, and D Sivakumar. 2004. An information statistics approach to data stream and communication complexity. *J. Comput. System Sci.* 68, 4 (2004), 702–732.

[5] Ran Ben Basat, Xiaoqi Chen, Gil Einziger, and Ori Rottenstreich. 2020. Designing heavy-hitter detection algorithms for programmable switches. *IEEE/ACM Transactions on Networking* 28, 3 (2020), 1172–1185.

[6] Radu Berinde, Piotr Indyk, Graham Cormode, and Martin J Strauss. 2010. Space-optimal heavy hitters with strong error bounds. *ACM Transactions on Database Systems (TODS)* 35, 4 (2010), 1–28.

[7] Arnab Bhattacharyya, Palash Dey, and David P Woodruff. 2018. An optimal algorithm for 1-heavy hitters in insertion streams and related problems. *ACM Transactions on Algorithms (TALG)* 15, 1 (2018), 1–27.

[8] Burton H Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.

[9] Robert S Boyer and J Strother Moore. 1991. MJRTY—a fast majority vote algorithm. In *Automated Reasoning*. Springer, 105–117.

[10] Mark Braverman, Sumegha Garg, and David P Woodruff. 2020. The coin problem with applications to data streams. In *2020 IEEE 61st Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 318–329.

[11] Vladimir Braverman, Stephen R Chestnut, Nikita Ivkin, Jelani Nelson, Zhengyu Wang, and David P Woodruff. 2017. BPTree: an 2 heavy hitters algorithm using constant memory. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 361–376.

[12] Moses Charikar, Kevin Chen, and Martin Farach-Colton. 2002. Finding frequent items in data streams. In *International Colloquium on Automata, Languages, and Programming*. Springer, 693–703.

[13] Graham Cormode and Marios Hadjieleftheriou. 2008. Finding frequent items in data streams. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1530–1541.

[14] Graham Cormode, Theodore Johnson, Flip Korn, Shan Muthukrishnan, Oliver Spatscheck, and Divesh Srivastava. 2004. Holistic UDAFs at streaming speeds. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. 35–46.

[15] Graham Cormode, Tejas Kulkarni, and Divesh Srivastava. 2019. Answering range queries under local differential privacy. *Proceedings of the VLDB Endowment* 12, 10 (2019), 1126–1138.

[16] Graham Cormode, Samuel Maddock, and Carsten Maple. 2021. Frequency Estimation under Local Differential Privacy [Experiments, Analysis and Benchmarks]. *arXiv preprint arXiv:2103.16640* (2021).

[17] Graham Cormode and Shan Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58–75.

[18] Graham Cormode and Shan Muthukrishnan. 2005. What's hot and what's not: tracking most frequent items dynamically. *ACM Transactions on Database Systems (TODS)* 30, 1 (2005), 249–278.

[19] Graham Cormode and Ke Yi. 2020. *Small Summaries for Big Data*. Cambridge University Press.

[20] Sudipto Das, Shyam Antony, Divyakant Agrawal, and Amr El Abbadi. 2009. Cots: A scalable framework for parallelizing frequency counting over data streams. In *2009 IEEE 25th International Conference on Data Engineering*. IEEE, 1323–1326.

[21] Erik D Demaine, Alejandro López-Ortiz, and J Ian Munro. 2002. Frequency estimation of internet packet streams with limited space. In *European Symposium on Algorithms*. Springer, 348–360.

[22] Min Fang, Narayanan Shivakumar, Hector Garcia-Molina, Rajeev Motwani, and Jeffrey D Ullman. 1999. Computing Iceberg Queries Efficiently.. In *Internaational Conference on Very Large Databases (VLDB'98), New York, August 1998*. Stanford InfoLab.

[23] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. 2007. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm.

[24] Anna C Gilbert, Yannis Kotidis, S Muthukrishnan, and Martin J Strauss. 2002. How to summarize the universe: Dynamic maintenance of quantiles. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*. Elsevier, 454–465.

[25] Michael Greenwald and Sanjeev Khanna. 2001. Space-efficient online computation of quantile summaries. *ACM SIGMOD Record* 30, 2 (2001), 58–66.

[26] Şule Gündüz and M Tamer Özsu. 2003. A web page prediction model based on click-stream tree representation of user behavior. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*. 535–540.

[27] Rob Harrison, Shir Landau Feibish, Arpit Gupta, Ross Teixeira, S Muthukrishnan, and Jennifer Rexford. 2020. Carpe elephants: Seize the global heavy hitters. In *Proceedings of the Workshop on Secure Programmable Network Infrastructure*. 15–21.

[28] Nikita Ivkin, Edo Liberty, Kevin Lang, Zohar Karnin, and Vladimir Braverman. 2019. Streaming Quantiles Algorithms with Small Space and Update Time. *arXiv preprint arXiv:1907.00236* (2019).

[29] Rajesh Jayaram and David P Woodruff. 2018. Data streams with bounded deletions. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 341–354.

[30] Cheqing Jin, Weining Qian, Chaofeng Sha, Jeffrey X Yu, and Aoying Zhou. 2003. Dynamically maintaining frequent items over a data stream. In *Proceedings of the twelfth international conference on Information and knowledge management*. 287–294.

[31] Hossein Jowhari, Mert Sağlam, and Gábor Tardos. 2011. Tight bounds for lp samplers, finding duplicates in streams, and related problems. In *Proceedings of the thirtieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 49–58.

[32] John Kallaugher and Eric Price. 2020. Separations and equivalences between turnstile streaming and linear sketching. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*. 1223–1236.

[33] Zohar Karnin, Kevin Lang, and Edo Liberty. 2016. Optimal quantile approximation in streams. In *2016 ieee 57th annual symposium on foundations of computer science (focs)*. IEEE, 71–78.

[34] Richard M Karp, Scott Shenker, and Christos H Papadimitriou. 2003. A simple algorithm for finding frequent elements in streams and bags. *ACM Transactions on Database Systems (TODS)* 28, 1 (2003), 51–55.

[35] Nishad Manerikar and Themis Palpanas. 2009. Frequent items in streaming data: An experimental evaluation of the state-of-the-art. *Data & Knowledge Engineering* 68, 4 (2009), 415–430.

[36] Gurmeet Singh Manku and Rajeev Motwani. 2002. Approximate frequency counts over data streams. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*. Elsevier, 346–357.

[37] Gurmeet Singh Manku, Sridhar Rajagopalan, and Bruce G Lindsay. 1998. Approximate medians and other quantiles in one pass and with limited memory. *ACM SIGMOD Record* 27, 2 (1998), 426–435.

[38] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. 2005. Efficient computation of frequent and top-k elements in data streams. In *International conference on database theory*. Springer, 398–412.

[39] Jayadev Misra and David Gries. 1982. Finding repeated elements. *Science of computer programming* 2, 2 (1982), 143–152.

[40] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. 2005. Interpreting the data: Parallel analysis with Sawzall. *Scientific Programming* 13, 4 (2005), 277–298.

[41] Nisheeth Shrivastava, Chiranjeeb Buragohain, Divyakant Agrawal, and Subhash Suri. 2004. Medians and beyond: new aggregation techniques for sensor networks. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*. 239–249.

[42] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, Shan Muthukrishnan, and Jennifer Rexford. 2017. Heavy-hitter detection entirely in the data plane. In *Proceedings of the Symposium on SDN Research*. 164–176.

[43] Daniel Ting. 2018. Data sketches for disaggregated subset sum and frequent item estimation. In *Proceedings of the 2018 International Conference on Management of Data*. 1129–1140.

[44] Lu Wang, Ge Luo, Ke Yi, and Graham Cormode. 2013. Quantiles over data streams: an experimental study. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 737–748.

[45] Victor Zakhary, Lawrence Lim, Divyakant Agrawal, and Amr El Abbadi. 2020. CoT: Decentralized elastic caches for cloud environments. *arXiv preprint arXiv:2006.08067* (2020).

[46] Yinda Zhang, Zaoxing Liu, Ruixin Wang, Tong Yang, Jizhou Li, Ruijie Miao, Peng Liu, Ruwen Zhang, and Junchen Jiang. 2021. CocoSketch: high-performance sketch-based measurement over arbitrary partial key query. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 207–222.

[47] Fuheng Zhao, Divyakant Agrawal, Amr El Abbadi, and Ahmed Metwally. 2021. SpaceSaving$^{\pm}$: An Optimal Algorithm for Frequency Estimation and Frequent items in the Bounded Deletion Model. arXiv:2112.03462 [cs.DB]

[48] Fuheng Zhao, Sujaya Maiyya, Ryan Wiener, Divyakant Agrawal, and Amr El Abbadi. 2021. KLL±: Approximate quantile sketches over dynamic datasets. *Proceedings of the VLDB Endowment* 14, 7 (2021), 1215–1227.

[49] George Kingsley Zipf. 2016. *Human behavior and the principle of least effort: An introduction to human ecology*. Ravenio Books.

ByteGNN: Efficient Graph Neural Network Training at Large Scale

Chenguang Zheng[1,2,†], Hongzhi Chen[2,*], Yuxuan Cheng[2], Zhezheng Song[1], Yifan Wu[2,3,†], Changji Li[1,2,†], James Cheng[1], Hao Yang[2], Shuai Zhang[2]
[1]{cgzheng, cjli, jcheng}@cse.cuhk.edu.hk, [1]szaizai18@gmail.com, [2]{zhengchenguang, chenhongzhi, chengyuxuan.911, wuyifan.18, lichangji, yanghao.2019, zhangshuai.root}@bytedance.com, [3]yifanwu@pku.edu.cn
[1]The Chinese University of Hong Kong, [2]ByteDacne Inc, [3]Peking University

ABSTRACT

Graph neural networks (GNNs) have shown excellent performance in a wide range of applications such as recommendation, risk control, and drug discovery. With the increase in the volume of graph data, distributed GNN systems become essential to support efficient GNN training. However, existing distributed GNN training systems suffer from various performance issues including high network communication cost, low CPU utilization, and poor end-to-end performance. In this paper, we propose ByteGNN, which addresses the limitations in existing distributed GNN systems with three key designs: (1) an abstraction of mini-batch graph sampling to support high parallelism, (2) a two-level scheduling strategy to improve resource utilization and to reduce the end-to-end GNN training time, and (3) a graph partitioning algorithm tailored for GNN workloads. Our experiments show that ByteGNN outperforms the state-of-the-art distributed GNN systems with up to 3.5-23.8 times faster end-to-end execution, 2-6 times higher CPU utilization, and around half of the network communication cost.

PVLDB Reference Format:
Chenguang Zheng, Hongzhi Chen, Yuxuan Cheng, Zhezheng Song, Yifan Wu, Changji Li, James Cheng, Hao Yang, and Shuai Zhang. ByteGNN: Efficient Graph Neural Network Training at Large Scale. PVLDB, 15(6): 1228 - 1242, 2022.
doi:10.14778/3514061.3514069

1 INTRODUCTION

Existing systems for neural network training, such as TensorFlow [5] and PyTorch [54], are designed for training euclidean data such as images, texts and audio data. In recent years, a new type of neural networks, called **graph neural networks** (**GNNs**), become popular because of the ubiquity of graph data today such as semantic web graphs, knowledge graphs, social networks, and e-commerce networks. GNNs combine the non-euclidean graph structures with traditional neural networks to extract rich information from graph data for machine learning. Recent research results [16, 31, 43, 64, 71, 75] have shown that GNNs achieve significant performance improvements over traditional methods on many important tasks such as node classification, link predication, and graph clustering. GNNs have been applied in a broad range of applications including recommendation systems [52, 75], computer vision [50, 58], natural language processing [55, 73], drug discovery [24], and social networks [68].

Although many graph computing systems [8, 9, 18, 22, 27, 42, 49, 51, 67, 72, 79, 81] have been proposed, they are designed for batch graph analytics workloads such as the computation of PageRank, shortest paths, label propagation and connected components, and thus they lack of operators for neural network training. Thus, dedicated GNN systems have been developed upon neural network training systems (e.g., TensorFlow, PyTorch) for GNN training.

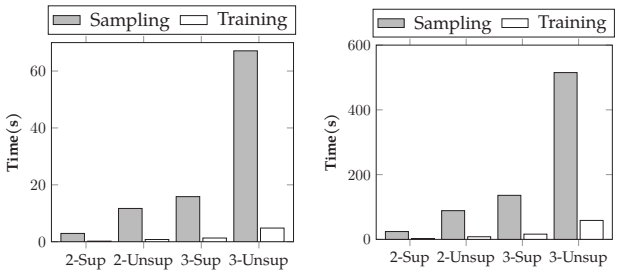
Among existing GNN systems, most of them are still single-machine systems, e.g., DGL [66], PyTorch Geometric (PyG) [21], NeuGraph [48], FeatGraph [33] and Seastar [70], which are optimized for training GNN models on a relatively small graph but cannot scale to process large graphs generally available in industry today. Note that for GNNs, a graph not only contains the graph topology information (which is typically used for computations such as PageRank, shortest paths, etc.), but each vertex and edge in the graph also contain a feature vector. Thus, depending on the dimensions of the feature vectors (typically around 100 to hundreds), the size of a graph for GNNs can be easily many times larger than the graph topology processed by existing graph computing systems. For example, for the Ogbn-Papers [32] graph used in our experiments, a feature vector has 128 dimensions and the size of the features is 4 times larger than the size of the graph topology.

For GNN training on large graphs, distributed systems such as Euler [1], GraphLearn (also called AliGraph) [80], AGL [77] and DistDGL [78] have been proposed. During the training, these systems collect and aggregate the feature vectors of the K-hop neighbors in order to compute the feature vector of each vertex, where K is the number of layers of the GNN model to be trained. However, the K-hop neighbors of a vertex can be many, especially for a power-law graph, and a large portion of them can be located in remote machines. Thus, fetching **all** the K-hop neighbors to a local machine for each vertex (referred to as **full mini-batch training**) incurs high network communication overheads and memory consumption.

To address the problem of full mini-batch training, **mini-batch sampling training** was proposed [13, 31, 34], which works as follows. Distributed GNN training is conducted in iterations and for each iteration, a machine processes a mini-batch of vertices in

* Hongzhi Chen is the Corresponding Author.
† This work was done when the authors were in ByteDance.
The work of Chenguang Zheng, Changji Li and James Cheng was partially supported by a ByteDance Research Collaboration Project.

Proceedings of the VLDB Endowment, Vol. 15, No. 6 ISSN 2150-8097.
doi:10.14778/3514061.3514069



(a) Reddit **(b) Ogbn-Product**

Figure 1: Sampling and training time of GraphLearn

its partition in two phases: (1) **the sampling phase** — for each vertex v in the mini-batch, the sampler samples a limited number of neighbors of v in each hop, fetches the sampled remote neighbors to the local machine, and constructs the neighborhood subgraph of v from its sampled neighbors locally; (2) **the training phase** — the trainer trains the model on the neighborhood subgraphs of the vertices in the mini-batch locally.

While distributed mini-batch sampling has become the default method for GNN training on a large graph (for which full-batch training and full mini-batch training are not practical), existing distributed GNN training systems suffer from a number of performance problems. One main problem is that sampling can take significantly longer time to complete than training, due to large amounts of random data access and remote data fetching involved in the sampling phase. For example, Figure 1 reports the average sampling time and training time in an epoch of training a 2-layered and 3-layered GraphSAGE model (both supervised and unsupervised) on four machines by GraphLearn [80] on the Reddit dataset [31] and Ogbn-Product dataset [32], which shows that the sampling phase takes an order of magnitude longer time than the training phase. For example, in the 2-layer supervised GraphSAGE training on Ogbn-Product, GraphLearn's training time is only 2.66s while its sampling time is 24.17s. Under the same setting, the sampling time of DistDGL [78] is also 4.22x of its training time.

The imbalance between the sampling and training phases also leads to the under-utilization of computing resources and the problem is worsen if GPUs are used for training (which further widens the gap between the sampling and training time) [59]. To address this imbalanced computing pattern in mini-batch GNN training, existing systems have attempted to apply neighborhood caching [46] and fixed size prefetching [78] to shorten the sampling time. However, it is difficult to set the right hyper-parameters (i.e., cache ratio and prefetching number) for training different GNN models on different graphs. Nextdoor [36] proposed to sample neighborhood using GPUs, but the GPU memory capacity limits the size of the graph it can handle. Graph partitioning has also been applied to reduce the cost of remote data fetching [80]. However, existing graph partitioning algorithms are designed for traditional graph workloads (e.g., distributed PageRank) and they do not consider the data access pattern and load balancing in GNN training.

In this paper, we propose ByteGNN, a distributed GNN training framework to support fast end-to-end GNN training in large graphs. To improve the efficiency of sampling, we abstract the sampling phase of a mini-batch as a directed acyclic graph (DAG) of small tasks, so that we can run DAGs and tasks within each DAG in parallel. The fine-grained task abstraction in DAG modeling also leads to the design of a two-level scheduling. First, coarse-grained scheduling determines how much resources should be used for mini-batch sampling, in order to dynamically adjust the computation loads between the sampling and training phases to avoid resource contention and maximize CPU utilization. Then, fine-grained scheduling decides the execution order of tasks in the DAGs in order to pipeline the sampling outputs to be consumed by the training phase at the right pace. The two scheduling strategies work together to minimize the end-to-end GNN training time. We also propose an effective graph partitioning algorithm tailored for mini-batch graph sampling, which maintains the data locality according to the data access pattern of mini-batch sampling and balances the computation loads in the training, validation and testing stages.

We implemented ByteGNN based on GraphLearn [4]. Our performance evaluation shows that ByteGNN achieves significantly higher training throughput and is more scalable than the state-of-the-art distributed GNN systems. Experimental results show that ByteGNN achieves up to 23.8x speedup over GraphLearn and 3.5x over DistDGL. The results verify that our system designs lead to efficient GNN training.

2 BACKGROUND AND MOTIVATION

We first introduce the necessary background of GNN and briefly discuss sampling-based GNN training. Then, we motivate our work by presenting the limitations of existing systems for large-scale GNN training.

2.1 Graph Neural Networks

GNN models are designed to capture the information contained in both the relationship among vertices in a graph and the vertex/edge attributes. The core idea of GNNs is recursively aggregating the neighbor information, including the features of the neighbors and the features of the connecting edges, and then applying feature transformation functions.

Take the GraphSAGE model [31] as an example. The training process for one layer of the model can be expressed as follows:

$$h^k_{\mathcal{N}(v)} \leftarrow AGGREGATE_k(\{h^{k-1}_u, \forall u \in \mathcal{N}(v)\}), \quad (1)$$

$$h^k_v \leftarrow \sigma(W^k \cdot CONCAT(h^{k-1}_v, h^k_{\mathcal{N}(v)})), \quad (2)$$

where $\mathcal{N}(v)$ is the set of neighbors of vertex v. In this k-th convolution layer, each vertex v first uses the *AGGREGATE* function to collect the feature vectors of v's neighbors from the $(k-1)$-th layer. The aggregation result is then concatenated with v's feature vector from the $(k-1)$-th layer, followed by a dot-product operation with a learnable weight matrix $\mathbf{W}$. The dot-product result is further transformed by a nonlinearity activation function σ such as the sigmoid function, which gives the feature vector of v for the k-th layer.

As the number of layers increases, the vertices are required to gather and aggregate more and more information from neighbors

that are farther away (i.e., expanding from the i-hop neighbors to the j-hop neighbors for $j > i$). When the training for all the K layers completes (for a user-specified K), the final feature vector h_v^K for each vertex v is fed into a mapping function for a specific downstream task (e.g., node classification, link prediction).

2.2 Distributed Mini-Batch Graph Sampling

Existing distributed GNN systems adopt data parallelism and sampling is commonly applied in order to train a GNN model on a large graph efficiently. However, the sampling process in distributed GNN training is quite different from that in training DNN models for computer vision and natural language processing, for which each sample is independent and small. For GNNs, the distributed training may access the entire neighborhood of a sampled vertex, including both vertices and edges along with their feature vectors. Due to the structural connection among vertices in different partitions, the data access pattern usually leads to high network communication cost.

In a K-layered GNN model, for each sampled *seed* vertex v, we need to obtain the K-hop neighborhood of v to construct a neighborhood subgraph to update v's feature vector. As most real-world graphs have a power-law degree distribution, the size of the K-hop neighborhood subgraph of a vertex grows exponentially as the number of hops increases. To address this problem, **mini-batch neighborhood sampling** [46, 78, 80] has been used to sample a limited number of neighbors for each sampled seed vertex. Figure 2 illustrates how mini-batch graph sampling is applied in the training of a 2-layered GNN model. We show the sampled 2-hop neighborhood subgraphs of two seed vertices, v_1 and v_2, where we set the sampling configuration $D_1 = 2$ and $D_2 = 2$, meaning that a vertex v first samples at most D_1 of its 1-hop neighbors, and then each u of v's sampled 1-hop neighbors further samples D_2 of u's neighbors. Remote sampling requests are sent to remote devices to access the i-hop neighbors that are stored there. After the sampling finishes, the sampled neighbors, along with their attributes (which are used to construct the initial feature vectors), are fetched to the local device of v_1 (and v_2) to construct its sampled 2-hop neighborhood subgraph, which is then fed into the GNN model to calculate the gradients and update the model parameters.

Although the tradeoff is a potential loss in the model accuracy, mini-batch graph neighborhood sampling still converges to the required model accuracy. Take the mean aggregator in GraphSAGE [31] as an example, using the Monte Carlo estimation, for the layer k, we obtain:

$$\mathbb{E}[h^k_{\mathcal{N}^s(v)}] = \mathbb{E}[\frac{1}{|\mathcal{N}^s(v)|}\sum_{u\in\mathcal{N}^s(v)} h_u^{k-1}] = \frac{1}{|\mathcal{N}(v)|}\sum_{u\in\mathcal{N}(v)} h_u^{k-1} = h^k_{\mathcal{N}(v)},$$

where $\mathcal{N}^s(v)$ is the set of random sampled neighbors of vertex v and $|\mathcal{N}^s(v)| = D_k$, which is the fanout of layer k. Unfortunately, though $h^k_{\mathcal{N}^s(v)}$ is an unbiased estimator of $h^k_{\mathcal{N}(v)}$, $h^k_{v^s}$ is not an unbiased estimator of h_v^k due to the non-linearity of $\sigma(\cdot)$ in Equation (2) [14]. Thus, the gradient is biased and the convergence of SGD is not guaranteed, unless the fanout D_k goes to infinity. But in practice, GraphSAGE sets $D_1 = 25$ and $D_2 = 10$ to provide statistically significant gains over existing approaches [43, 56]. In addition, AGL [77] also reported that with a suitable sample size, the sample can well approximate the ground truth.

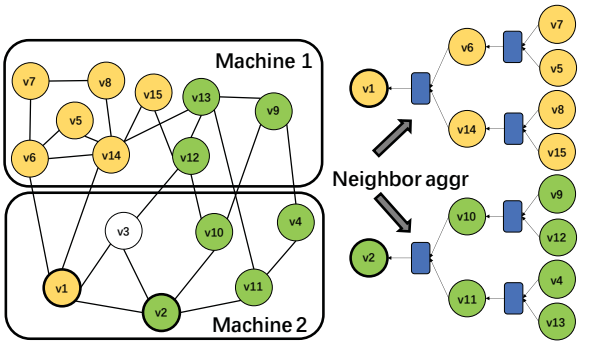


Figure 2: 2-hop mini-batch graph sampling

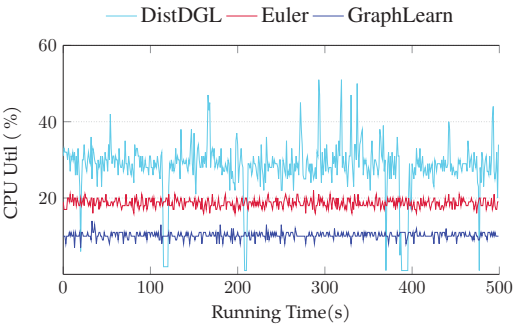


Figure 3: The CPU utilization of different systems

2.3 Limitations of Existing GNN Systems

Existing systems for distributed GNN training suffer from the following major limitations.

(1) The overhead of network communication is large. As the sampling procedure shows in Section 2.2, for every sampled seed vertex v in each iteration, we need to construct v's K-hop neighborhood subgraph together with the feature vectors of the vertices and edges in the subgraph. For example, in a 3-hop neighborhood subgraph where the sampling configuration is set at $k_1 = 15$, $k_2 = 10$ and $k_3 = 5$ in the Reddit dataset [31], there are 915 vertices each with a 602-dimension feature vector, which are what we need to prepare for one sampled seed vertex. As many of the neighbors and their features may be stored in remote machines, the K-hop neighborhood subgraph construction incurs a high network communication overhead. Figure 13(a) shows that the number of remote vertices is about six times that of local vertices with the widely used Hash partitioning. In fact, existing graph partitioning algorithms only consider to reduce the inter-partition edges, but do not consider the data access pattern and load balancing of graph sampling in GNN training. This calls for a new design of a more effective graph partitioning strategy tailored for GNN training.

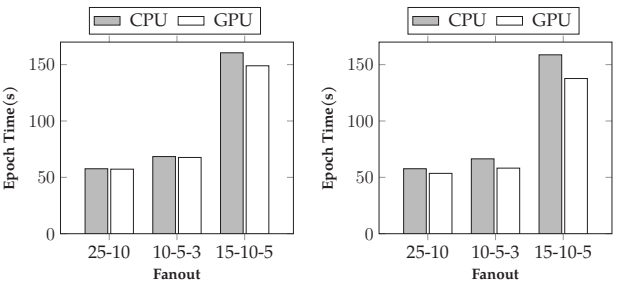
(2) CPU utilization is low. Our performance profiling shows that existing distributed GNN systems had poor CPU utilization as shown in Figure 3. By analyzing their system designs, we list the main causes to their low CPU utilization below.

The sampling phase of GraphLearn [80] is handled using the Gremlin semantics [2, 3] to express each sampling step. For each step, the Gremlin statement is translated by a parser and converted into several execution operations. An operation is a minimum execution unit in GraphLearn. GraphLearn has low CPU utilization since all the graph sampling operations within each device do not overlap with each other. DistDGL [78] takes a similar approach but also has many optimizations such as replicating the neighbors of its local vertices.

Euler [1], on the other hand, wraps each graph operator into one TensorFlow dataflow operator. This design is convenient for users to build the whole computation graph in TensorFlow. However, as all the sampling operators and the training operators are contained in one big computation dataflow graph, existing deep learning systems (including TensorFlow) cannot process it efficiently. It is difficult to have the optimal execution order for the dataflow graph with the newly defined graph sampling operators, which is totally different from the normal tensor computation. Besides, due to the convergence requirement, TensorFlow only runs one dataflow graph at a time. Each iteration always starts graph sampling after the previous training process finishes. This design also eliminates the opportunities to apply the data prefetching mechanism to the independent sampling stages.

(3) GPU does not bring enough benefit for GNN training in large graphs. As mentioned in Section 1, distributed GNN training on large graphs consists of the sampling phase and training phase. Due to limited GPU memory capacity, graph data are stored in the host memory of the machines and thus the sampling phase is conducted by CPUs. When GPUs are used to conduct the training phase, the sampling results are loaded into GPU memory from CPU host memory via PCIe links. As shown in [46, 59], even in the case of single-machine GNN training using GPU (i.e., data are not fetched through network), the sampling and data loading time still take a significant portion of the end-to-end training time. The training phase can indeed be accelerated using GPUs (compared with using CPUs), but this only reduces the model updating time while the sampling phase still dominates the overall processing cost. This is because most GNN models are considerably small (unlike DNN models) and the training phase only needs to conduct model computation on densely packed vectors, while sampling a large graph involves large amounts of random data access and remote data fetching in order to construct the neighborhood subgraph for each sampled seed vertex.

We evaluated the performance of DistDGL [78] on a GPU server (40 cores Intel(R) Xeon(R) Silver 4114 CPU @ 2.20GHz, 256 GB Memory, and one Nvidia RTX 2080Ti graphics card). We tested DistDGL with different fanout and hidden sizes to show the influence from the workloads of sampling and training. As Figure 4 shows, the largest difference in epoch time is only 10% between using GPU and using CPU. The average GPU utilization for GNN training is only around 20%, which is consistent with the GPU utilization of DGL reported in [46]. In fact, even if we purely use CPUs for the training phase, the sampling phase still dominates the overall cost as we have shown in Figure 1. In addition, we also need to consider the operational costs. GPU servers are expensive and GPU quota is more restricted to training DNNs even in big companies like ByteDance. In the cloud environment, Dorylus [61] also shows that



(a) hidden size 32 **(b) hidden size 256**

Figure 4: The epoch time of DistDGL with different hidden sizes and fanout on the Ogbn-product dataset

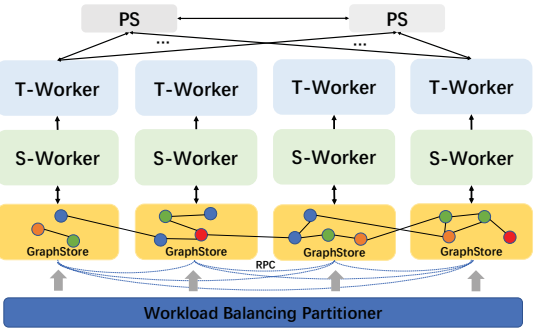


Figure 5: System architecture of ByteGNN

GPU-based training is only cost-effective for small, dense graphs. All the above concerns therefore divert our focus to designing a CPU-based framework for large-scale GNN training.

Motivation summary. The analysis above motivates us to design (1) a computation paradigm that uses only CPUs and aims to maximize CPU utilization by adaptively allocating computing resources to the sampling and training phases according to their needs, and (2) a new graph partitioning algorithm in order to reduce massive network communication caused by graph sampling in GNN training.

3 SYSTEM DESIGN

Figure 5 shows the architecture of ByteGNN, which consists of four main components in each machine where ByteGNN is deployed. **Graph Store** stores a partition of the input graph data and the Graph Stores of all machines form a distributed Graph Store. **PS** is a parameter server that stores the model parameters. **Sampling Worker** (**S-Worker**), handles the sampling phase and constructs sampled neighborhood subgraphs for sampled seed vertices. **Training Worker** (**T-Worker**), handles the training phase, which computes model gradients on the sampled neighborhood subgraphs constructed by the S-Worker in the same machine and synchronizes the gradients with PS to update the model parameters.

In Sections 3.1-3.3 we focus on three key designs in ByteGNN, which address the limitations of existing GNN systems discussed in Section 2.3.

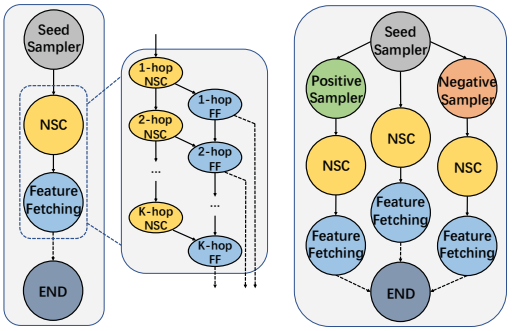


Figure 6: The DAG of the sampling workflow

3.1 Abstraction of Mini-Batch Graph Sampling

The sampling process in existing GNN systems [21, 78, 80] is not well-organized as the tasks in each sampling phase are executed without overlapping, which often leads to CPU under-utilization. In addition, sampling is conducted for one iteration (i.e., one mini-batch) after another, even though different mini-batches are independent of each other. To support parallel sampling within a mini-batch and among mini-batches, so as to maximize CPU utilization, we model the sampling process as a DAG of tasks. Then, we can execute the DAGs of sampling multiple mini-batches in parallel. We also introduce a scheduler in Section 3.2 to effectively utilize the computing resources for both intra-DAG and inter-DAG parallelization, while balancing the loads between sampling and training so that one is not waiting for the other to finish in order to continue.

To construct this DAG for a general GNN model, we analyzed the sampling phase of a broad range of existing GNN models, i.e., those that follow a similar neighborhood aggregation as described in Section 2.1, which cover most of the widely-adopted models such as GCN [43], GAT [64], GraphSAGE [31], PinSAGE [75], and GraphSAINT [76]. We provide a common abstraction for the sampling phase of these GNN models with a set of five operators: (1) **Seed Sampler**: sampling a set of vertices as *seeds* from the local graph store; (2) **Positive Sampler**: sampling vertices from the direct neighbors of each seed; (3) **Negative Sampler**: sampling vertices from those that are not the direct neighbors of each seed; (4) **Neighborhood Subgraph Construction (NSC)**: sampling vertices from the multi-hop neighborhood of a given vertex and constructing the sampled neighborhood subgraph; (5) **Feature Fetching**: fetching the attributes of a given vertex/edge to construct its feature vector.

With the above five operators, we can present the workflow of the sampling phase as a DAG, as shown in Figure 6. The DAG on the left of Figure 6 models supervised training, which consists of three tasks: Seed Sampling, NSC, and Feature Fetching. For unsupervised training, we also need to construct the neighborhood subgraphs of each positively and negatively sampled vertices of the seed vertices, as shown in the DAG on the right of Figure 6. The three branches in the DAG for unsupervised training can be executed in parallel, and the results are then collected in the "End" node to be fed into a T-Worker for training.

To enable higher parallelism for both supervised and unsupervised training, we create an instance of the two dominating operations (i.e., NSC and Feature Fetching, as they access multi-hop neighbors and their attributes) for each sampled vertex and execute these instances in parallel. In addition, as NSC (along with Feature Fetching) is executed repeatedly for each hop of neighborhood expansion, we can break the multi-hop operations into many smaller tasks of one-hop operations. As shown in Figure 6, each small task of Feature Fetching can start immediately when the corresponding small NSC task finishes. The more fine-grained task abstraction results in higher parallelism and better resource utilization (e.g., less head-of-line blocking and stragglers, less fragmentation in resource utilization).

To construct a DAG, users only need to specify the customized sampling functions in Seed Sampler, Positive Sampler, Negative Sampler, and also in NSC (e.g., how and how many neighbors in each hop should be sampled). This design also leaves space for researchers and engineers to explore new, high-quality sampling strategies using the framework. Note that the logical DAG is created only once and physical instances are generated and executed for each mini-batch by the S-Workers.

3.2 Two-Level Scheduling

ByteGNN adopts a two-level scheduling strategy to improve CPU utilization and reduce the end-to-end GNN training time. Although many scheduling strategies have been proposed, they are mostly for job scheduling at the cluster level [17, 19, 20, 25, 28–30, 35, 40, 47, 57, 60, 63, 74] or heterogeneous jobs/tasks in dataflow systems [39], which are over-complicated and incur extra overheads for scheduling the simple tasks in our system (note that for the training of a GNN model, we only need to schedule instances of the same DAG instead of many different DAGs).

Coarse-grained scheduling. The S-Worker in each machine executes multiple DAGs in parallel to increase throughput and reduce the end-to-end GNN training time. The first question we need to answer is how many DAGs should be launched in a machine. If we launch too many DAGs, which means more resource is needed by sampling, then resource contention becomes a problem. Resource contention does not just occur among the DAGs, but also between sampling (i.e., DAG execution) and training (i.e., model computation). The training time increases significantly when too many DAGs are launched. On the other hand, if too few DAGs are running, the resource is under-utilized. The training phase finishes quickly and the next iteration's training waits for the neighborhood subgraphs to be produced by the DAGs.

To control the resource utilization, we need to decide when to launch a DAG. We can model this problem as a variation of the classical Job-Shop Scheduling Problem (JSP) [7]. Each DAG can be regarded as a job, where a set of operations (tasks) in each job need to be processed in a specific order, and we have a set of jobs that are to be processed on a given set of workers. Knowing the best timing for DAG launching is equivalent to getting the earliest starting time of each job in the solution to this special Job-Shop Scheduling Problem. The Job-Shop Scheduling Problem has been well studied and to find a schedule that minimizes the makespan or minimizes the sum of the job completion time was

Algorithm 1: The Coarse-Grained Scheduling Strategy

Variable: C_{util}, Q_{size}, T_{gap}
Given: σ=*launch-score*
while *more_dag* **do**
  //*more_dag*=1 when more DAGs can be launched
  $balance = \frac{T_{avg\_sample}}{T_{avg\_train} * Q_{size}}$;
  $f(C_{util}) = (101 - e^{C_{util}/c})$, where $c = \frac{100}{\ln_{101}}$;
  launch-score = T_{gap} * $f(C_{util})$ * $balance$;
  if *launch-score* $\geq \sigma$ **then**
    more_dag = *Launch_DAG()*;
    // launches a new DAG; returns 0 when no more DAG to launch
    $T_{last\_launch} = Time()$ // used to calculate T_{gap}
  else
    sleep(5ms);
  end
end

proved to be strongly NP-hard [11]. Some new research also shows that the currently best approximation algorithms have worse than logarithmic performance guarantee [26].

We propose a heuristic strategy to decide when to launch a DAG based on three runtime measures: C_{util}, Q_{size}, and T_{gap}.

C_{util} is the CPU utilization rate. If C_{util} is low, we may launch a new DAG; otherwise, we may wait until C_{util} drops to a suitable level. Note that high C_{util} does not necessarily result in better performance because there could be much contention and switching among DAGs and between sampling and training.

In addition to CPU utilization, We also need to consider the memory footprint. The neighborhood subgraph constructed from each DAG execution is kept in the DAG output queue in the S-Worker and Q_{size} is the size of this queue. The neighborhood subgraphs are then consumed by the T-Worker for training. Thus, Q_{size} is essentially an indicator of the speed of production (by the S-Worker) and the speed of consumption (by the T-Worker) of the neighborhood subgraphs. If Q_{size} is small, we may launch new DAGs; otherwise, we pause the launching. If Q_{size} is large, it implies an over-supply of neighborhood subgraphs and we may shift more computing resource from sampling to accelerate training. Thus, Q_{size} not only controls the memory usage, but also balances the overall resource usage between sampling and training.

We also found that the real-time measure for C_{util} is not sensitive enough since newly launched DAGs may not change the CPU utilization in a short time period and many DAGs may be launched during the period. Later, when the tasks in these DAGs start to run in parallel and use up the computing resource, the system suffers from severe resource contention. To avoid such delayed performance punishments, we introduce T_{gap}, which is the time gap elapsed since the previous DAG launch. If T_{gap} is too small, we may want to wait for a bit longer before we launch a new DAG.

It would be undesirable if users need to set the thresholds for the three measures, as it is hard to determine what values of C_{util}, Q_{size}, and T_{gap} are good and how to relate them to each other. To this end, we integrate them into one single score, *launch-score*, to decide whether we should launch a new DAG. The idea is to maintain the balance between the production speed and the consumption speed of neighborhood subgraphs, while keeping CPU utilization high. Ideally, we hope that the output of each DAG will be consumed immediately by the training phase, which means that Q_{size} should be close to 0 all the time. However, in most of the cases a very low Q_{size} happens with a very low C_{util}. Thus, we need to consider Q_{size} together with C_{util}.

Algorithm 1 shows the algorithm for coarse-grained scheduling. First, we want to maintain $balance = \frac{T_{avg\_sample}}{T_{avg\_train} * Q_{size}} = 1$, where T_{avg_sample} and T_{avg_train} are the average time for sampling and training a mini-batch. If $balance > 1$, it means that it would take less time to consume the current Q_{size} sampling results than to produce a new sampling result, which is an indicator that a new DAG should be launched. Next, we first attempt to use $(100 - C_{util})$ to give a higher weight to *balance* if C_{util} is low and penalize *balance* (i.e., delay new DAG launching) when C_{util} is high. However, simply using $(100 - C_{util})$ does not work well as it is a linear scale. Instead, we want to quickly increase CPU utilization when C_{util} is low and prevent contention promptly when C_{util} is already very high. Thus, we use an exponential function, $f(C_{util}) = 101 - e^{C_{util}/c}$, where $c = \frac{100}{ln101}$ is a constant used to align the range of $f(C_{util})$ with that of C_{util}, i.e., $f(0) = 100$, $f(100) = 0$, and $0 \leq f(C_{util}) \leq 100$. Finally, we also put T_{gap} as a weight to reflect the delay in the real-time measurement of C_{util}, which leads to the definition of *launch-score* in Algorithm 1.

We monitor *launch-score* in real time and launch a new DAG when *launch-score* $\geq \sigma$, where σ is a threshold set as follows. As shown in Algorithm 1 and explained above, *launch-score* connects *balance*, $f(C_{util})$ and T_{gap} together to determine whether a new DAG job should be launched. In practice, there exist reasonable values of *balance*, $f(C_{util})$ and T_{gap} for which a new DAG should be launched; Note that there are always trade-offs between *balance* and $f(C_{util})$, e.g., a higher *balance* and a lower $f(C_{util})$, to achieve a high *launch-score*. Such tradeoffs in runtime allows the system to automatically adjust the resource allocation to balance the sampling and training progress.

Fine-grained scheduling. After new DAGs are launched, the S-Worker executes the tasks in the DAGs, in parallel with the tasks in other DAGs. These tasks are put in a queue when their dependency is cleared (i.e., their parent tasks in the DAG are completed) and are handled by a pool of processing threads. If we execute the tasks in an FIFO order, some tasks of newly launched DAGs could be in front of the tasks in those almost-finished DAGs. For example, when the DAG_1 pushes the "END" node in the task queue and there are already "NSC" tasks from DAG_2 and DAG_3 in the queue, the "NSC" tasks will be executed first and the "END" task will be processed later even although the "END" task is the last task in DAG_1, completing which will immediately return the sampled data to the T-Worker for training. Meanwhile, one task may unlock a lot of downstream tasks in the same DAG, and heavy tasks may block many light tasks. Thus, the average completion time of the DAG jobs and hence the end-to-end GNN training time can be significantly increased.

We schedule tasks according to the following orders: (1) tasks in a DAG with a smaller ID will be executed first; (2) tasks in the same DAG will be executed in ascending order of their costs. We assign a smaller ID to a DAG launched earlier to prioritize earlier DAGs to be

Algorithm 2: Block Assignment

Input: List of Blocks $\mathbf{B} = B_1, B_2, ..., B_n$
Output: Graph partitions $P_1, P_2, P_3, ..., P_k$
for *each block* B_i *in* $\boldsymbol{B}$ **do**
  for $j \leftarrow 1$ *to* k **do**
    CE[j] = |Cross_Edge(P_j, B_i)| / $|P_j|$
    BS[j] = $(1\text{-}\alpha * \frac{|P_j(train)|}{C(train)}\text{-}\beta * \frac{|P_j(val)|}{C(val)}\text{-}\gamma * \frac{|P_j(test)|}{C(test)})$
  end
  x = **argmax** $_{1 \le t \le k}$ {CE[t] * BS[t]}
  $P_x = P_x \cup B_i$
end
return $P_1, P_2, P_3, ..., P_k$

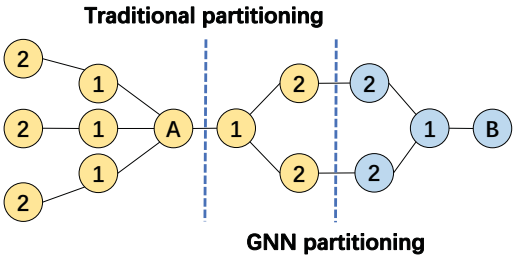


Figure 7: Traditional partitioning vs. GNN partitioning

completed first. We calculate the cost of a task by the data it needs to handle. For example, for sampling tasks in each hop of NSC, the cost is equal to the total number of neighbors of the input vertices; for Feature Fetching, the cost is the number of vertices/edges to be fetched multiplied by the vertex/edge feature dimension. As tasks may require data from remote machines, the S-Worker sends data fetching requests to the local Graph Store, which communicates with remote Graph Stores to fetch the data. The remote requests are also scheduled in a similar way and the network operations are processed concurrently with the CPU operations.

3.3 GNN-based Graph Partitioning

Existing graph partitioning algorithms [37, 41, 46] are mainly designed to reduce inter-partition edges and balance the workload. They have been widely adopted in distributed graph processing systems [27, 53, 81] to reduce inter-machine communication. However, sample-based GNN training focuses on the K-hop neighborhood of only the vertices in the *training*, *validation* and *test* sets (instead of all vertices). For example, in Figure 7, traditional partitioning strategies cut the graph into two parts by the left dotted line since it not only balances the vertices but also has the least cut edge. But for a 2-layer GNN training, since vertex A and vertex B are the labeled vertices, partitioning by the right dotted line is actually a better choice. Even if this results in two cut edges, it would not cause any data movement in the training process as only the 2-hop neighbors of the labeled vertices are required.

In addition, the ratio of the sizes of the training, validation, and test sets of different real-world graphs may differ significantly. For example, in the Ogbn-Product dataset, the test set size is 11 times the training set size and 56 times the validation set size; while in the Ogbn-Papers dataset, the test set size is only 0.18 times the training set size and 1.7 times the validation set size. Thus, the partitioning algorithm should consider both the special data access pattern of k-layer GNN training and the balanced distribution of the training, validation, and test sets.

It is known that the traditional graph partitioning problem is proved to be APX-hard [6]. Thus, our graph partitioning problem is also APX-hard as it can be reduced to the traditional graph partitioning problem. We propose a heuristic two-step graph partitioning strategy tailored for GNN sampling workloads. The main idea is to group vertices into multi-hop neighborhood-based blocks and then assign these blocks to partitions by balancing the numbers of *training*, *validation* and *test* vertices in the partitions.

Step (1) neighborhood block construction. To better preserve the locality of graph data for GNN sampling workloads, we construct a neighborhood block for each vertex in the training, validation and test sets. We start a K-hop breadth-first search from each vertex v (v is called the **block center**) and broadcast the unique block ID of v to its K-hop neighbors being visited. Every vertex only keeps the first block ID it receives, except for block centers which keep their own block ID. A block is then formed of all the vertices that keep the same block ID. Figure 7 demonstrates how to construct the blocks.

Step (2) block assignment. Just as existing graph partitioning algorithms aim to balance the number of vertices in the partitions, our objective is to also balance the number of training, validation and test vertices in the partitions so that the work of training, validation and test is also balanced among the machines. Algorithm 2 shows how to assign the blocks. For each block B_i, it is assigned to the partition with the highest score. P_j is the set of vertices that have already been assigned to partition j. $CE[j]$ is the number of cross-edges between B_i and P_j, which will be eliminated if B_i is assigned to P_j. Thus, the larger $CE[j]$ is, the more likely B_i is assigned to P_j. As the size of different partitions may vary during the assignment, we normalize $CE[j]$ by $|P_j|$. $BS[j]$ is the balancing score that controls the number of training/validation/test vertices in partition j to be close to the average value. For example, the expected number of training vertices in each partition is $C(train) = |V(train)|/N$, where $V(train)$ is the set of all training vertices and N is the total number of partitions. Let $P_j(train)$ be the set of training vertices currently in partition j. Thus, a smaller $\frac{|P_j(train)|}{C(train)}$ means that more training vertices can be assigned to partition j. The above applies to the validation and test vertices as well. In addition, we also use a weight to put more attention on a specific type of vertices according to the scale of that type in order to obtain a better overall performance. For example, if the number of training vertices is significantly more, we may set a larger α to favor the training process, which can improve the end-to-end processing time.

Before the block assignment, we sort the blocks in descending order of $max\{|V(train)|, |V(val)|, |V(test)|\}$. Then, we start the block assignment according to this order. In this way, larger blocks are assigned to different partitions first, so that smaller blocks may

be used later to fill the partitions more easily when the partitions begin to fill up.

4 SYSTEM IMPLEMENTATION

We implemented ByteGNN based on GraphLearn [4], using TensorFlow [5] as the backend deep learning framework for the training phase. We used the data loader and distributed graph storage in GraphLearn, where the graph topology data is stored in adjacency list format and the features are stored separately and indexed by their vertex/edge ID. Our implementation focuses on efficient DAG construction and execution, graph partitioning, and gradient synchronization.

DAG construction and execution. We adopt the Gremlin syntax to help us construct the DAG. We redesigned the parsing method to encode necessary metadata from a Gremlin query for generating DAG nodes. Since one Gremlin statement may become several nodes in the final DAG, we implemented the parsing phase to carefully handle the complex dependency among the task nodes. We also changed all the communication methods from synchronous in GraphLearn to asynchronous in ByteGNN.

Graph partitioning. We implemented our graph partitioning strategy on the streaming graph partitioning framework in [12, 46]. The random start seed vertices in [12] were replaced with labeled vertices/edges. The framework first does the multi-source distributed BFS to build the K-hop neighborhood blocks, and then applies our block assignment strategy in Section 3.3 to assign blocks to the partitions. The partitions are written into HDFS and then loaded by the system for sampling and training.

Gradient synchronization. To address the potential convergence issue, we implemented the bulk synchronous parallel (BSP) and stale synchronous parallel (SSP) models based on the TensorFlow API, so that users may also choose to use BSP or SSP to obtain faster model convergence and reduce the training time.

5 SYSTEM EVALUATION

We evaluate the performance of ByteGNN by comparing with Graph-Learn [4], Euler [1] and Distributed DGL (DistDGL) [78]. We also examine the effects of our system designs on the performance.

Testbed. We ran our experiments on a cluster of machines where each machine is equipped with 1T DDR4 main memory and two 2.40GHz Intel(R) Xeon(R) Platinum 8260 CPU (each CPU has 24 cores or 48 virtual cores by hyper-threading). All the machines are connected by a 25Gbps network and the OS is the Debian 9.13 with Linux kernel 4.19.117.

Datasets. We used three datasets in the evaluation, as shown in Table 1. *Ogbn-Product* and *Obgn-Papers* are the largest two graphs in the Open Graph Benchmark (OGB). *Ogbn-Product* is an undirected and unweighted graph modeling an Amazon product co-purchasing network [32]. *Obgn-Papers* is a directed citation graph of 111 million papers indexed by MAG [65]. The *Social* dataset is a directed graph in industry from the social network scenario.

Models. We used three representative GNN models, Graph Convolutional Network (GCN) [43], GraphSAGE [31] and Graph Attention network(GAT) [64], in our evaluation. In order to demonstrate the expressiveness and efficiency of ByteGNN, we also tested the unsupervised variants of these three models. Although unsupervised learning shares most of the GNN architectures with supervised learning, it involves the negative sampling operator in the sampling phase and is also widely used in important tasks such as link prediction. Since many works are proposed to improve the sampling of GNN models, we used GraphSAINT [76] as a typical example to show how our sampling abstraction can be applied.

Table 1: Graph datasets

Dataset	Ogbn-Product (Product)	Ogbn-Papers (Papers)	Social
Vertices	2,449,029	111,059,956	66,351,656
Edges	123,718,280	1,615,685,872	1,751,915,191
Feature	100	128	150
Classes	47	172	2
Training set	196,615	1,207,179	6,631,989
Validation set	39,323	125,265	19,908,461
Test set	2,213,091	214,338	39,811,206

As shown by prior works [16, 32, 37], deeper and larger GNN architectures can achieve better model accuracy. We used three network layers for the models and set the sampling configuration to $k_1 = 10$, $k_2 = 5$ and $k_3 = 3$ for the neighborhood sampling models. The mini-batch size was set to 512 in all the experiments.

Systems. We compared with three distributed GNN training systems, GraphLearn, Euler (v1.0) and DistDGL (DGL v0.5.3). GraphLearn is a distributed framework designed for the development and application of GNNs on large scale graphs within Alibaba. Euler is also developed by Alibaba but it has been used in many companies for large scale GNN training. Both GraphLearn and Euler use TensorFlow as the backend system. DistDGL is a popular GNN system and its latest version (v0.5.3) supports distributed GNN training. The computational patterns of DistDGL are highly optimized by dedicated sparse tensor operations, which are currently lacking in ByteGNN as this work focuses on improving the sampling performance. Unless otherwise stated, we used the default configuration of these systems in our experiments. ByteGNN used the BSP model to obtain better test accuracy. All the systems adopt the random neighborhood sampling method as the default sampling method and use the same hop number and fanout.

5.1 Overall Performance

We first compared the overall performance of the systems. We report the throughput of each system, i.e., the number of samples being processed per second, which is a metric commonly used to measure the performance of model training of a system. The throughput is calculated as the total number of seed vertices processed divided by the end-to-end GNN training time. Thus, the larger the throughput of a system, the shorter is the end-to-end GNN training time of the system. The hidden size is set to 32 in GCN and GraphSAGE. For GAT, we used 4 attention heads with hidden size 16. Since Euler failed to run unsupervised GAT training, we ignore this result.

Figure 8 reports the results. ByteGNN achieves 7.5 to 16.2 times speedup compared with GraphLearn on supervised training and up to 23 times on unsupervised training. As ByteGNN is implemented on GraphLearn and the key differences from GraphLearn are the three system designs presented in Sections 3.1-3.3, the results show

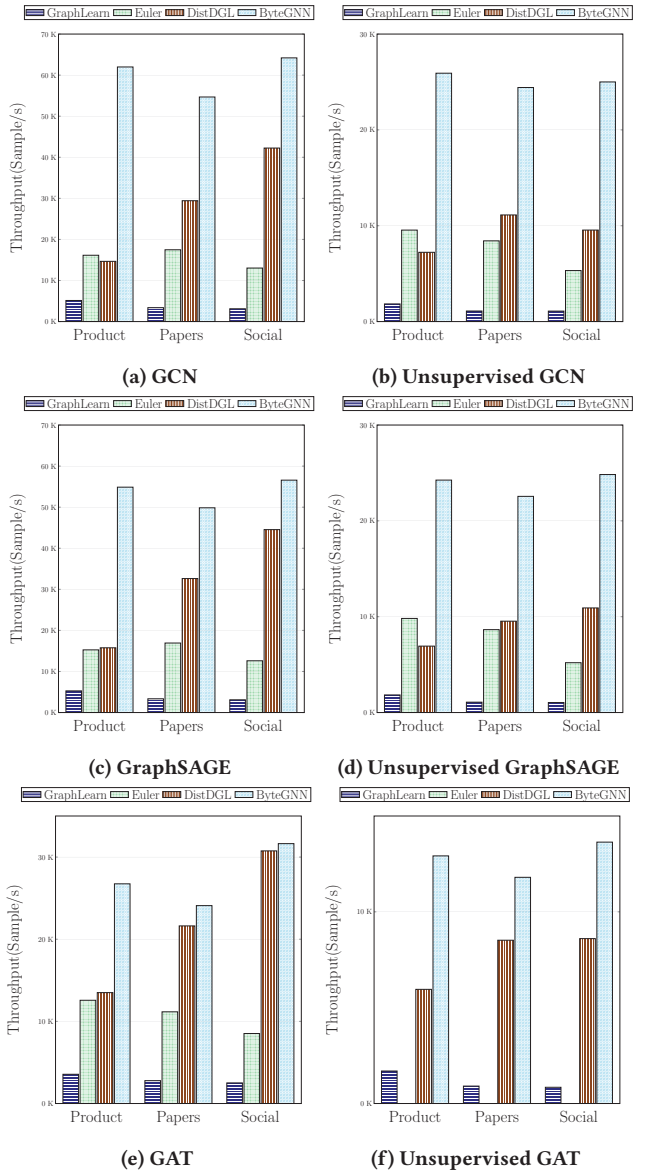


Figure 8: The throughput of GraphLearn, Euler, DistDGL, and ByteGNN for training different models on 4 machines

that our designs are effective. In particular, the performance improvement obtained by ByteGNN is more significant for unsupervised training that has more parallel sampling tasks, which is as a result of the high parallelism enabled for tasks within a DAG and among DAGs.

Compared with Euler, although Euler also adopts the data-flow graph by TensorFlow for sampling the mini-batch neighborhood and training, ByteGNN can still achieve up to 4.7 times performance speedup. Euler cannot run two TensorFlow's computation graphs at the same time as otherwise it would lead to a convergence problem. In contrast, the separation of sampling phase and training phase in ByteGNN enables concurrent execution of multiple DAGs to maximize CPU utilization.

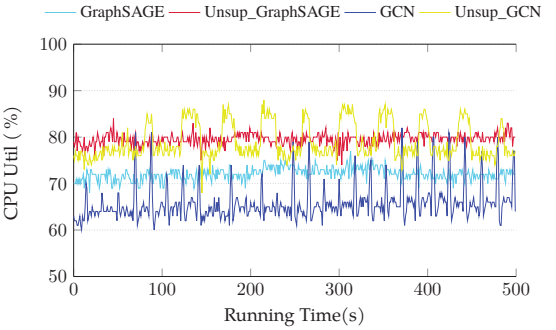


Figure 9: Average CPU utilization of ByteGNN

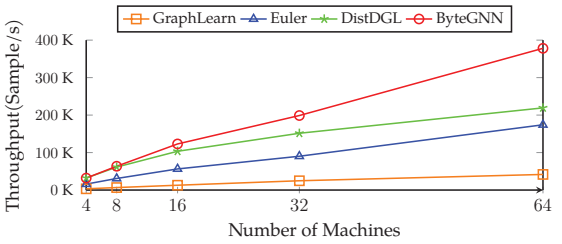
Compared with DistDGL, ByteGNN achieves 2.1 ~ 3.5 times speedup for training the dense graph *Ogbn-Product* in both supervised and unsupervised training. For the sparse graphs *Ogbn-Papers* and *Social*, ByteGNN still has better performance. In the supervised training, ByteGNN is 1.5x and 1.3x faster than DistDGL in GCN and GraphSAGE. But the speedup is less significant compared with that on the dense graph, especially for GAT. This is because sparse tensor operations in the training phase of DistDGL have been highly optimized, while currently there is no such optimization in ByteGNN. For unsupervised training that has heavier sampling workloads, Figures 8(b)&(d)&(f) show that ByteGNN achieves considerably better performance as ByteGNN's design enables higher parallelism in sampling execution. (e.g., 2.4x for unsupervised GraphSAGE and 1.6x for unsupervised GAT).

We also report the average CPU utilization of ByteGNN for training all the models in Figure 9. The result is reported for *Ogbn-Papers*, while ByteGNN's CPU utilization for the other two datasets is similar. Compared with the average CPU utilization of GraphLearn, Euler and DistDGL as shown in Figure 3, ByteGNN achieves 3 - 6 times higher CPU utilization. ByteGNN has lower CPU utilization for supervised GCN and GraphSAGE because the number of neighborhood subgraphs in the DAG output queue is sufficient, S-Worker dynamically frees up some resource to T-Worker and the training workload for GCN is not heavy.

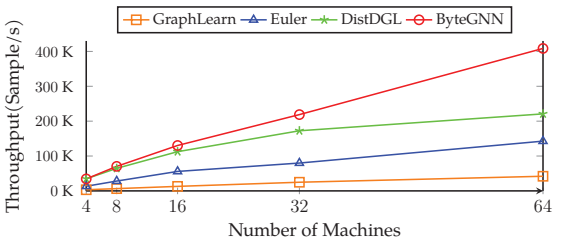
5.2 Scalability

Figure 10 reports the throughput scalability of the systems for the *Ogbn-Papers* dataset, where we increase the number of machines from 4 to 64. ByteGNN achieves better scalability than all the other three systems. We omit the results for the other two datasets due to the page limitation, but the patterns are similar and ByteGNN's performance on the dense graph *Ogbn-Product* is even better. The hidden size is set to 256 here to demonstrate the performance of our system in different configuration.

In general, the throughput performance in distributed GNN training has sub-linear scalability due to the synchronization overhead (when the BSP model is used to achieve high accuracy) and heavy network I/O among the machines. GraphLearn and Euler scale poorly and their throughputs are relatively low. Although GraphLearn and Euler are built on top of TensorFlow, the default asynchronous gradient update in distributed TensorFlow does not



(a) GraphSAGE



(b) GCN

Figure 10: Scalability comparison

cause much synchronization overhead (though with potential accuracy loss). However, without an effective graph partitioning algorithm to preserve the locality of neighborhood access, remote data fetching results in high network communication overhead. In contrast, DistDGL's main issue in scalability is due to the synchronization overhead for gradient update. If the sampling output of a mini-batch cannot return on time, the trainer will get the forward loss later and all the other machines will wait for this loss to begin the back propagation. Even with the fixed prefetching mechanism, the possibility of the back propagation waiting increases as the number of machines increases. In comparison, ByteGNN's scheduling allows the sampling outputs to be pipelined to the trainers while other sampling processes continue, which results in better resource utilization. ByteGNN's GNN-tailored graph partitioning algorithm also leads to lower network communication overhead as the number of machines increases. As a result, ByteGNN achieves better scalability than the other systems.

5.3 Model Accuracy

We also report the correctness of ByteGNN by evaluating the test accuracy of the GraphSAGE model on the *Ogbn-Product* dataset, comparing with GraphLearn and DistDGL. Euler has similar accuracy as GraphLearn. In Figure 11, we report the test accuracy of different systems at every epoch until the training converges. The result shows that the systems achieve similar or the same accuracy eventually, but ByteGNN converges the fastest, in both the single-machine setting (1M) and distributed 4-machine setting (4M). We also note that as the mini-batch training can update the model many times in one epoch, the accuracy increases quickly in the first several epochs. The single-machine accuracy of GraphLearn can also be seen as the baseline to demonstrate that our code changes to GraphLearn do not affect the semantics of the GNN models. And as ByteGNN uses BSP to ensure model convergence in distributed training, it achieves approximately the same accuracy as DistDGL but uses less time.

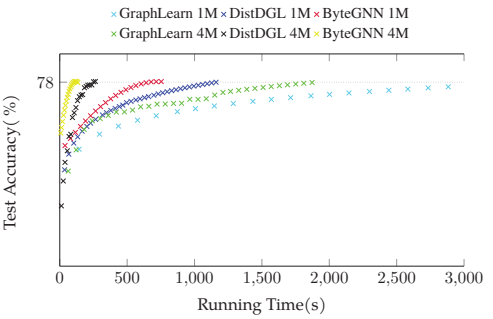


Figure 11: Accuracy comparison

5.4 Evaluation on System Designs

We further evaluate the effectiveness of each individual system design in ByteGNN.

5.4.1 Sampling Abstraction. We used the GraphSAINT model to demonstrate how to build a DAG using our sampling abstraction. Different from sampling neighbors across the layers, GraphSAINT constructs mini-batches by sampling the whole input graph once and then building a full GCN on the sampled subgraph. It provides three light-weight and efficient samplers, NodeSampler, EdgeSampler, and RandomWalkSampler. Due to space constraints, we only show the implementation of the Seed Sampler function in our sampling abstraction for GraphSAINT's NodeSampler and EdgeSampler. Note that the training part is the same for different samplers.

```
// NodeSampler
def seed_sampler(self):
  return self.g.V(node_type="train")
                .batch(batch_size = n)
                .by("InDegree")

// EdgeSampler
def seed_sampler(self):
  return self.g.E(edge_type="train")
                .batch(batch_size = m)
                .by("EdgeWeight").bothV()
```

For GraphSAINT's NodeSampler, we sample n vertices from all the training vertices according to a vertex probability distribution $P(u) \propto ||\tilde{A}_{:,u}||^2$. We call this "InDegree" sampling as it is associated with the in-degree of each vertex. For EdgeSampler, the edge probability distribution follows $p_{e(v,u)} \propto \frac{1}{deg(u)} + \frac{1}{deg(v)}$. Normally, it can be pre-calculated dependent on the graph topology only and become the weight of edges. The code above shows the Seed Sampler function of these two samplers using our sampling abstraction. Using Gremlin, users can easily write the sampling logic. Then, the sampling stage can be completed by the NSC and Feature Fetching functions as discussed in Section 3.1.

We also implemented the GraphSAINT model in GraphLearn to compare the end-to-end training performance. Table 2 reports the speedup ratio of ByteGNN over GraphLearn for training GraphSAINT on different graphs using four machines, using the same sampling setting from [76]. Even though GraphSAINT has a light sampling workload, ByteGNN can still achieve significant speedup compared with GraphLearn. Note that ByteGNN has better performance with EdgeSampler because EdgeSampler needs to obtain the two end-vertices of the sampled edge and has a higher workload than NodeSampler.

Table 2: Speedup ratio of ByteGNN over GraphLearn

Type	Ogbn-Product	Ogbn-Papers	Social
NodeSampler	3.40	2.80	4.05
EdgeSampler	4.72	3.25	5.89

Table 3: The execution time (sec) of one epoch for different sampling settings, running on *Ogbn-Papers* using 8 machines

(a) The execution time of light sampling workload

	Sequential	Fixed DAGs	Coarse-Grained
512	79.04	19.56	18.62
1024	75.29	19.21	17.52
2048	74.52	19.90	17.75

(b) The execution time of heavy sampling workload

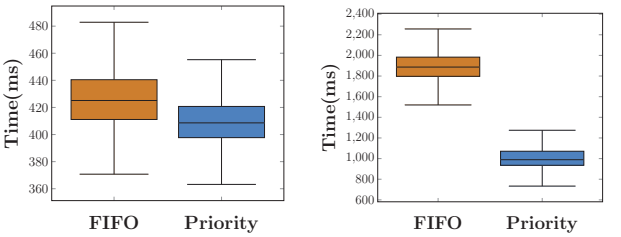
	Sequential	Fixed DAGs	Coarse-Grained
512	314.04	63.41	56.70
1024	312.14	68.72	57.20
2048	310.43	78.10	62.46

5.4.2 Coarse-Grained Scheduling. We first evaluate the performance of the coarse-grained scheduling strategy. We used three different batch sizes: 512, 1024 and 2048. We created a light sampling workload by setting the sampling configuration to $k_1 = 10$, $k_2 = 5$ and $k_3 = 3$. We also created a heavy sampling workload by setting the sampling configuration to $k_1 = 15$, $k_2 = 10$ and $k_3 = 5$.

We used two baselines. The first baseline is sequential DAG execution, which runs DAGs one after another. The second baseline is running a fixed number of DAGs at any time. The DAG size is set to 16 which is the same as the default in DistDGL prefetching.

Table 3 shows that coarse-grained scheduling achieves the best performance in all cases. For sequential DAG execution, the execution of a single DAG at a time results in resource under-utilization and thus has poor performance. For the light sampling workload, the fixed number of DAGs has performance close to that of coarse-grained scheduling. This is because the sampling workload is light and can be finished quickly so that DAGs can already produce the sampling results fast enough for the trainer to consume. However, the lower sampling rate leads to more biased results and the light workload also results in resource under-utilization. When the sampling workload is heavier, the higher random data access overhead and higher network I/O cost to retrieve remote neighbors become the performance bottleneck. In this case, our coarse-grained scheduling strategy becomes effective as it dynamically adjusts the resource allocation to sampling and training in order to maximize resource utilization.

5.4.3 Fine-Grained Scheduling. We further show the impact of the fine-grained scheduling strategy on the DAG completion time. We ran ByteGNN for 10 epochs and measured the completion time of each DAG of mini-batch sampling under two settings: using the priority-based scheduling in the fine-grained scheduling strategy and using FIFO-based scheduling. We use the box plot to report the distribution of the DAG completion time. Figure 12 shows that the priority-based scheduling can significantly reduce the completion time of the DAGs, and the DAG completion time is also more stable, which avoids the short time period of under-supply or over-supply of the sampling outputs. In supervised training, when the number of DAGs is in the suitable range, there are not too many small tasks in the DAGs so that the FIFO scheduling can handle it. However, unsupervised training launches more sampling tasks during the DAG execution. The median DAG completion time of the FIFO scheduling is almost two times greater than the median of the priority-based scheduling.



(a) Supervised Training **(b) Unsupervised Training**

Figure 12: Distribution of DAG completion time

5.4.4 Graph Partitioning. To validate the effectiveness of our GNN-tailored graph partitioning (**GNN-P**) algorithm, we compared GNN-P with three well-known graph partitioning methods: hash partitioning, Fennel partitioning [62] and METIS partitioning [41]. Both hash and METIS partitioning have been widely adopted in distributed graph computing systems. Fennel is the representative of one-pass streaming partitioning algorithms.

Figure 13 reports the distribution of the requests for remote and local neighborhood data in one training epoch by each machine (the distributions of the requests for validation and test show a similar pattern). First, Figure 13(a) shows that hash partitioning achieves the best balanced distribution because hash partitioning assigns each type of vertices to different partitions with the same possibility. However, it does not consider the locality of neighborhood data access and thus incurs much higher remote data requests, which result in high network communication overhead. The number of remote requests is about 6.32 times the local data requests. Although METIS keeps the total number of vertices similar in each partition, the number of training vertices varies significantly among the partitions (also true for validation and test vertices). Half of the training vertices are assigned to one partition in Machine 1, which indeed reduces remote data requests; however, the imbalanced distribution results in Machine 1 being a severe straggler, which processes around 80% of the data requests in each training epoch. Fennel roughly balances the total load in each partition. Fennel considers data locality but it is only limited to direct neighbors, and thus remote data requests still take a major portion of the total number of data requests in each partition. In contrast, the multi-hop block construction of GNN-P significantly improves the data locality of each partition. The ratio of remote data requests and local data requests in the partitions of GNN-P is also considerably

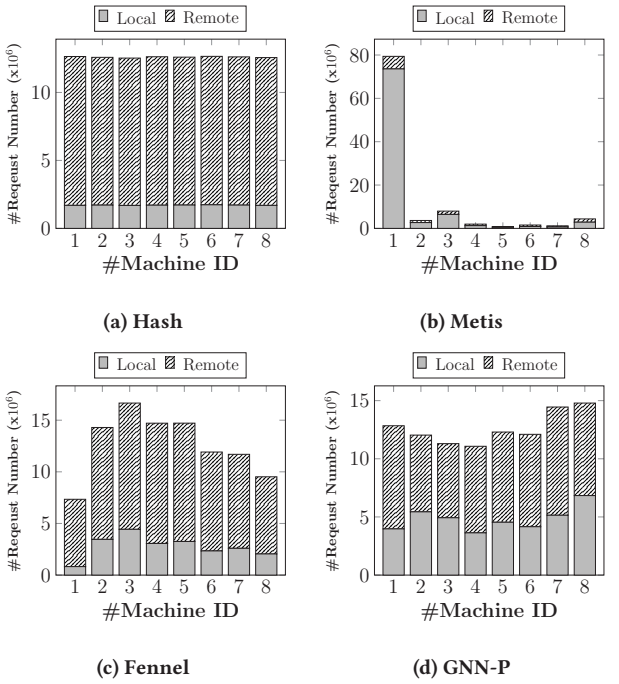


Figure 13: The distribution of remote/local data requests

smaller than that of the hash and Fennel partitions. In addition, with the balance-aware assignment algorithm, GNN-P achieves a much more balanced distribution of the total workload.

6 RELATED WORK

Single-machine GNN systems. PyG [21] integrates with PyTorch [54] to provide a message-passing API for GNN training. Incorporated with the Apache TVM compiler [15], FeatGraph [33] generates optimized kernels for both CPU and GPU. But to implement new GNN operators, users need to have the background of TVM primitives. NeuGraph [48] proposes Scatter-ApplyEdge-Gather-ApplyVertex programming model to express GNN models and supports full-batch training in a single machine with multiple GPUs. It divides a graph into 2-D chunks and introduces a streaming scheduler to handle the CPU-GPU data transfer when GNN computation for a graph cannot fit in the GPU memory. Seastar [69, 70] proposes a vertex-centric programming model to express GNN models using native Python syntax and identifies a common seastar computation pattern in GNN training to generate high-performance fused kernels. There are also works [36, 46] that focus on addressing the bottleneck of mini-batch sampling. PaGraph [46] is a sampling-based training framework on multi-GPUs that addresses the expensive subgraph data loading issue by a GNN computation-aware cache policy. NextDoor [36] enables users to express the sampling tasks in GPUs by a high-level API and also proposes a novel *transit parallelism* approach to parallelize graph sampling. However, these single-machine systems have the limitation of processing large industrial graphs due to limited GPU memory.

Distributed GNN systems. For training GNNs on large graphs in a distributed manner, AliGraph [80] provides sampling-based distributed GNN training and reduces network communication by caching vertices on local machines. DistDGL [78] uses a distributed in-memory key-value store to support efficient access to graph topology and feature data in distributed GNN training. DGCL [10] proposes an efficient communication library for distributed full-batch GNN training on multi-GPUs using NVLink. DGCL needs to load all the graph data into GPUs first and is not suitable for training large graphs. Based on FlexFlow [38], a distributed DNN training framework, Roc [37] also adopts full-batch training in multi-GPUs using dynamic programming to minimize data swapping between host DRAM and GPU memory. P3 [23] reduces communication by model parallelism for the first layer, while it uses data parallelism for the remaining layers. However, if the hidden size is larger than the input dimension, it still incurs a high cost for the synchronization of the output of the first layer. AGL [77] uses MapReduce to preprocess a graph, which samples multiple neighborhood subgraphs for each vertex and stores them in a distributed file system. During training, AGL loads the required samples of neighborhood subgraphs of the vertices directly from the disk. However, the preprocessing cost is high and the storage overhead can also be large. Dorylus [61] designs a computation separation mechanism and pipelines the different computation patterns in the Amazon EC2 machine and serverless Lambdas in the cloud environment.

Graph partitioning in GNN. METIS [41] is commonly used for graph partitioning in GNN algorithms [16, 44, 45] and systems [10, 48, 78]. Cluster-GCN [16] adopts METIS to build small clusters and then uses the partitions to perform an SGD update. DistDGL [78] adjusts the METIS algorithm to balance the training vertices in each partition. NeuGraph [48] uses the Kernighan-Lin algorithm to make the chunks in the diagonal have as many edges as possible. Roc [37] uses an linear-regression based algorithm to achieve balanced partitioning for both GNN training and inference; but it still treats all the vertices equally, which makes the computation load unbalanced. PaGraph [46] partitions a graph based on the neighborhood of a training vertex. However, with the multi-hop feature cache to avoid feature communication between different trainers, the memory overhead is too high.

7 CONCLUSIONS

We presented ByteGNN, a distributed GNN training system for GNN training in large graphs. ByteGNN abstracts the sampling phase of a mini-batch as a DAG of small tasks to support high parallelism. Leveraging the DAG abstraction, ByteGNN designs a two-level scheduling to improve resource utilization and reduce the end-to-end GNN training time. ByteGNN also tailors graph partitioning for GNN workloads to reduce network I/O and balance the workload. Experimental results show that ByteGNN can significantly shorten the end-to-end training time compared with existing distributed GNN systems.

ACKNOWLEDGMENTS

We thank the anonymous VLDB reviewers, for their constructive comments and suggestions that have helped greatly improve the quality of the paper.

REFERENCES

[1] 2019. Euler. https://github.com/alibaba/euler.

[2] 2019. Gremlin. http://tinkerpop.apache.org/gremlin.html.

[3] 2019. TinkerPop. http://tinkerpop.apache.org/.

[4] 2020. GraphLearn. https://github.com/alibaba/graph-learn.

[5] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. USENIX Association, 265–283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[6] Konstantin Andreev and Harald Räcke. 2004. Balanced graph partitioning. In *SPAA 2004: Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures, June 27-30, 2004, Barcelona, Spain*. ACM, 120–124. https://doi.org/10.1145/1007912.1007931

[7] David L. Applegate and William J. Cook. 1991. A Computational Study of the Job-Shop Scheduling Problem. *INFORMS J. Comput.* 3, 2 (1991), 149–156. https://doi.org/10.1287/ijoc.3.2.149

[8] Ching Avery. 2011. Giraph: Large-scale graph processing infrastructure on hadoop. *Proceedings of the Hadoop Summit. Santa Clara* 11 (2011).

[9] Ulrik Brandes. 2001. A faster algorithm for betweenness centrality. *The Journal of Mathematical Sociology* 25, 2 (2001), 163–177. https://doi.org/10.1080/0022250X.2001.9990249 arXiv:https://doi.org/10.1080/0022250X.2001.9990249

[10] Zhenkun Cai, Xiao Yan, Yidi Wu, Kaihao Ma, James Cheng, and Fan Yu. 2021. DGCL: an efficient communication library for distributed GNN training. In *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*. ACM, 130–144. https://doi.org/10.1145/3447786.3456233

[11] Bo Chen, Chris Potts, and Gerhard Woeginger. 1998. *A Review of Machine Scheduling: Complexity, Algorithms and Approximability*. 21–169. https://doi.org/10.1007/978-1-4613-0303-9_25

[12] Hongzhi Chen, Miao Liu, Yunjian Zhao, Xiao Yan, Da Yan, and James Cheng. 2018. G-Miner: an efficient task-oriented graph mining system. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*. ACM, 32:1–32:12. https://doi.org/10.1145/3190508.3190545

[13] Jie Chen, Tengfei Ma, and Cao Xiao. 2018. FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=rytstxWAW

[14] Jianfei Chen, Jun Zhu, and Le Song. 2018. Stochastic Training of Graph Convolutional Networks with Variance Reduction. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018 (Proceedings of Machine Learning Research)*, Vol. 80. PMLR, 941–949. http://proceedings.mlr.press/v80/chen18p.html

[15] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*. USENIX Association, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[16] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. 2019. Cluster-GCN: An Efficient Algorithm for Training Deep and Large Graph Convolutional Networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2019, Anchorage, AK, USA, August 4-8, 2019*. ACM, 257–266. https://doi.org/10.1145/3292500.3330925

[17] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles*. ACM, 153–167.

[18] Andrew A. Davidson, Sean Baxter, Michael Garland, and John D. Owens. 2014. Work-Efficient Parallel GPU Methods for Single-Source Shortest Paths. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, May 19-23, 2014*. IEEE Computer Society, 349–359. https://doi.org/10.1109/IPDPS.2014.45

[19] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, Vol. 48. ACM, 77–88.

[20] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: resource-efficient and QoS-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, Vol. 42. ACM, 127–144.

[21] Matthias Fey and Jan Eric Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. *CoRR* abs/1903.02428. arXiv:1903.02428 http://arxiv.org/abs/1903.02428

[22] Zhisong Fu, Bryan B. Thompson, and Michael Personick. 2014. MapGraph: A High Level API for Fast Development of High Performance Graph Analytics on GPUs. In *Second International Workshop on Graph Data Management Experiences and Systems, GRADES 2014, co-loated with SIGMOD/PODS 2014, Snowbird, Utah, USA, June 22, 2014*. CWI/ACM, 2:1–2:6. https://doi.org/10.1145/2621934.2621936

[23] Swapnil Gandhi and Anand Padmanabha Iyer. 2021. P3: Distributed Deep Graph Learning at Scale. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*. USENIX Association, 551–568. https://www.usenix.org/conference/osdi21/presentation/gandhi

[24] Thomas Gaudelet, Ben Day, Arian R. Jamasb, Jyothish Soman, Cristian Regep, Gertrude Liu, Jeremy B. R. Hayter, Richard Vickers, Charles Roberts, Jian Tang, David Roblin, Tom L. Blundell, Michael M. Bronstein, and Jake P. Taylor-King. 2020. Utilising Graph Machine Learning within Drug Discovery and Development. *CoRR* abs/2012.05716 (2020). arXiv:2012.05716 https://arxiv.org/abs/2012.05716

[25] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert NM Watson, and Steven Hand. 2016. Firmament: Fast, centralized cluster scheduling at scale. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*. 99–115.

[26] Leslie Ann Goldberg, Mike Paterson, Aravind Srinivasan, and Elizabeth Sweedyk. 1997. Better Approximation Guarantees for Job-shop Scheduling. In *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, 5-7 January 1997, New Orleans, Louisiana, USA*. ACM/SIAM, 599–608. http://dl.acm.org/citation.cfm?id=314161.314395

[27] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*. USENIX Association, 17–30. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gonzalez

[28] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. 2015. Multi-resource packing for cluster schedulers. In *Proceedings of the ACM SIGCOMM Computer Communication Review*, Vol. 44. ACM, 455–466.

[29] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. 2016. Altruistic scheduling in multi-resource clusters. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*. 65–80.

[30] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. 2016. GRAPHENE: Packing and Dependency-Aware Scheduling for Data-Parallel Clusters. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*. 81–97.

[31] William L. Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*. 1024–1034. https://proceedings.neurips.cc/paper/2017/hash/5dd9db5e033da9c6fb5ba83c7a7ebea9-Abstract.html

[32] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open Graph Benchmark: Datasets for Machine Learning on Graphs. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*. https://proceedings.neurips.cc/paper/2020/hash/fb60d411a5c5b72b2e7d3527cfc84fd0-Abstract.html

[33] Yuwei Hu, Zihao Ye, Minjie Wang, Jiali Yu, Da Zheng, Mu Li, Zheng Zhang, Zhiru Zhang, and Yida Wang. 2020. FeatGraph: A Flexible and Efficient Backend for Graph Neural Network Systems. *CoRR* abs/2008.11359 (2020). arXiv:2008.11359 https://arxiv.org/abs/2008.11359

[34] Wen-bing Huang, Tong Zhang, Yu Rong, and Junzhou Huang. 2018. Adaptive Sampling Towards Fast Graph Representation Learning. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*. 4563–4572. https://proceedings.neurips.cc/paper/2018/hash/01eee509ee2f68dc6014898c309e86bf-Abstract.html

[35] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. 2009. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*. ACM, 261–276.

[36] Abhinav Jangda, Sandeep Polisetty, Arjun Guha, and Marco Serafini. 2021. Accelerating graph sampling for graph machine learning using GPUs. In *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*. ACM, 311–326. https://doi.org/10.1145/3447786.3456244

[37] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2020. Improving the Accuracy, Scalability, and Performance of Graph Neural Networks with Roc. In *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*. mlsys.org. https://proceedings.mlsys.org/book/300.pdf

[38] Zhihao Jia, Matei Zaharia, and Alex Aiken. 2019. Beyond Data and Model Parallelism for Deep Neural Networks. In *Proceedings of Machine Learning and Systems 2019, MLSys 2019, Stanford, CA, USA, March 31 - April 2, 2019*. mlsys.org. https://proceedings.mlsys.org/book/265.pdf

[39] Tatiana Jin, Zhenkun Cai, Boyang Li, Chengguang Zheng, Guanxian Jiang, and James Cheng. 2020. Improving resource utilization by timely fine-grained scheduling. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*. ACM, 20:1–20:16. https://doi.org/10.1145/3342195.3387551

[40] Prajakta Kalmegh and Shivnath Babu. 2019. MIFO: A Query-Semantic Aware Resource Allocation Policy. In *Proceedings of the 2019 ACM International Conference on Management of Data*. ACM, 1678–1695.

[41] George Karypis and Vipin Kumar. 1998. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.* 20, 1 (1998), 359–392. https://doi.org/10.1137/S1064827595287997

[42] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. 2014. CuSha: vertex-centric graph processing on GPUs. In *The 23rd International Symposium on High-Performance Parallel and Distributed Computing, HPDC'14, Vancouver, BC, Canada - June 23 - 27, 2014*. ACM, 239–252. https://doi.org/10.1145/2600212.2600227

[43] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=SJU4ayYgl

[44] Ming Li, Zheng Ma, Yu Guang Wang, and Xiaosheng Zhuang. 2020. Fast Haar Transforms for Graph Neural Networks. *Neural Networks* 128 (2020), 188–198. https://doi.org/10.1016/j.neunet.2020.04.028

[45] Zhiheng Li, Geemi P. Wellawatte, Maghesree Chakraborty, Heta A. Gandhi, Chenliang Xu, and Andrew D. White. 2020. Graph Neural Network Based Coarse-Grained Mapping Prediction. *CoRR* abs/2007.04921 (2020). arXiv:2007.04921 https://arxiv.org/abs/2007.04921

[46] Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, and Yinlong Xu. 2020. PaGraph: Scaling GNN training on large graphs via computation-aware caching. In *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*. ACM, 401–415. https://doi.org/10.1145/3419111.3421281

[47] Libin Liu and Hong Xu. 2018. Elasecutor: Elastic Executor Scheduling in Data Analytics Systems. In *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 107–120.

[48] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. 2019. NeuGraph: Parallel Deep Neural Network Computation on Large Graphs. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*. USENIX Association, 443–458. https://www.usenix.org/conference/atc19/presentation/ma

[49] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *SIGMOD*. 135–146. https://doi.org/10.1145/1807167.1807184

[50] Kenneth Marino, Ruslan Salakhutdinov, and Abhinav Gupta. 2017. The More You Know: Using Knowledge Graphs for Image Classification. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*. IEEE Computer Society, 20–28. https://doi.org/10.1109/CVPR.2017.10

[51] Duane Merrill, Michael Garland, and Andrew S. Grimshaw. 2012. Scalable GPU graph traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2012, New Orleans, LA, USA, February 25-29, 2012*. ACM, 117–128. https://doi.org/10.1145/2145816.2145832

[52] Federico Monti, Michael M. Bronstein, and Xavier Bresson. 2017. Geometric Matrix Completion with Recurrent Multi-Graph Neural Networks. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*. 3697–3707. http://papers.nips.cc/paper/6960-geometric-matrix-completion-with-recurrent-multi-graph-neural-networks

[53] Anil Pacaci, Alice Zhou, Jimmy Lin, and M. Tamer Özsu. 2017. Do We Need Specialized Graph Databases? Benchmarking Real-Time Social Networking Applications. In *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems, GRADES@SIGMOD/PODS 2017, Chicago, IL, USA, May 14 - 19, 2017*. ACM, 12:1–12:7. https://doi.org/10.1145/3078447.3078459

[54] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*. 8024–8035. https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html

[55] Hao Peng, Jianxin Li, Yu He, Yaopeng Liu, Mengjiao Bao, Lihong Wang, Yangqiu Song, and Qiang Yang. 2018. Large-Scale Hierarchical Text Classification with Recursively Regularized Deep Graph-CNN. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web, WWW 2018, Lyon, France, April 23-27, 2018*. ACM, 1063–1072. https://doi.org/10.1145/3178876.3186005

[56] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. DeepWalk: online learning of social representations. In *The 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14, New York, NY, USA - August 24 - 27, 2014*. ACM, 701–710. https://doi.org/10.1145/2623330.2623732

[57] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. 2020. Autopilot: workload autoscaling at Google. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*. ACM, 16:1–16:16. https://doi.org/10.1145/3342195.3387524

[58] Victor Garcia Satorras and Joan Bruna Estrach. 2018. Few-Shot Learning with Graph Neural Networks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=BJj6qGbRW

[59] Marco Serafini. 2021. Scalable Graph Neural Network Training: The Case for Sampling. *ACM SIGOPS Oper. Syst. Rev.* 55, 1 (2021), 68–76. https://doi.org/10.1145/3469379.3469387

[60] Xiaoyang Sun, Chunming Hu, Renyu Yang, Peter Garraghan, Tianyu Wo, Jie Xu, Jianyong Zhu, and Chao Li. 2018. ROSE: Cluster Resource Scheduling via Speculative Over-Subscription. In *2018 IEEE 38th International Conference on Distributed Computing Systems*. IEEE, 949–960.

[61] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2021. Dorylus: Affordable, Scalable, and Accurate GNN Training with Distributed CPU Servers and Serverless Threads. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*. USENIX Association, 495–514. https://www.usenix.org/conference/osdi21/presentation/thorpe

[62] Charalampos E. Tsourakakis, Christos Gkantsidis, Bozidar Radunovic, and Milan Vojnovic. 2014. FENNEL: streaming graph partitioning for massive scale graphs. In *Seventh ACM International Conference on Web Search and Data Mining, WSDM 2014, New York, NY, USA, February 24-28, 2014*. ACM, 333–342. https://doi.org/10.1145/2556195.2556213

[63] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A Kozuch, Mor Harchol-Balter, and Gregory R Ganger. 2016. TetriSched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proceedings of the 11th European Conference on Computer Systems*. ACM, 35.

[64] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=rJXMpikCZ

[65] Kuansan Wang, Zhihong Shen, Chiyuan Huang, Chieh-Han Wu, Yuxiao Dong, and Anshul Kanakia. 2020. Microsoft Academic Graph: When experts are not enough. *Quant. Sci. Stud.* 1, 1 (2020), 396–413. https://doi.org/10.1162/qss_a_00021

[66] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, Ziyue Huang, Qipeng Guo, Hao Zhang, Haibin Lin, Junbo Zhao, Jinyang Li, Alexander J. Smola, and Zheng Zhang. 2019. Deep Graph Library: Towards Efficient and Scalable Deep Learning on Graphs. *CoRR* abs/1909.01315 (2019). arXiv:1909.01315 http://arxiv.org/abs/1909.01315

[67] Yangzihao Wang, Andrew A. Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D. Owens. 2016. Gunrock: a high-performance graph processing library on the GPU. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, March 12-16, 2016*. ACM, 11:1–11:12. https://doi.org/10.1145/2851141.2851145

[68] Zhouxia Wang, Tianshui Chen, Jimmy S. J. Ren, Weihao Yu, Hui Cheng, and Liang Lin. 2018. Deep Reasoning with Knowledge Graph for Social Relationship Understanding. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*. ijcai.org, 1021–1028. https://doi.org/10.24963/ijcai.2018/142

[69] Yidi Wu, Yuntao Gui, Tatiana Jin, James Cheng, Xiao Yan, Peiqi Yin, Yufei Cai, Bo Tang, and Fan Yu. 2021. Vertex-Centric Visual Programming for Graph Neural Networks. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 2803–2807. https://doi.org/10.1145/3448016.3452770

[70] Yidi Wu, Kaihao Ma, Zhenkun Cai, Tatiana Jin, Boyang Li, Chengguang Zheng, James Cheng, and Fan Yu. 2021. Seastar: vertex-centric programming for graph neural networks. In *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*. ACM, 359–375. https://doi.org/10.1145/3447786.3456247

[71] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How Powerful are Graph Neural Networks?. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. https://openreview.net/forum?id=ryGs6iA5Km

[72] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. 2014. Blogel: A Block-Centric Framework for Distributed Computation on Real-World Graphs. *Proc. VLDB*

Endow. 7, 14 (2014), 1981–1992. https://doi.org/10.14778/2733085.2733103

[73] Liang Yao, Chengsheng Mao, and Yuan Luo. 2019. Graph Convolutional Networks for Text Classification. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019*. AAAI Press, 7370–7377. https://doi.org/10.1609/aaai.v33i01.33017370

[74] Yi Yao, Han Gao, Jiayin Wang, Ningfang Mi, and Bo Sheng. 2016. OpERA: opportunistic and efficient resource allocation in Hadoop YARN by harnessing idle resources. In *Proceedings of the 25th International Conference on Computer Communication and Networks*. IEEE, 1–9.

[75] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. 2018. Graph Convolutional Neural Networks for Web-Scale Recommender Systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2018, London, UK, August 19-23, 2018*. ACM, 974–983. https://doi.org/10.1145/3219819.3219890

[76] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor K. Prasanna. 2020. GraphSAINT: Graph Sampling Based Inductive Learning Method. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net. https://openreview.net/forum?id=BJe8pkHFwS

[77] Dalong Zhang, Xin Huang, Ziqi Liu, Jun Zhou, Zhiyang Hu, Xianzheng Song, Zhibang Ge, Lin Wang, Zhiqiang Zhang, and Yuan Qi. 2020. AGL: A Scalable System for Industrial-purpose Graph Machine Learning. *Proc. VLDB Endow.* 13, 12 (2020), 3125–3137. https://doi.org/10.14778/3415478.3415539

[78] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. 2020. DistDGL: Distributed Graph Neural Network Training for Billion-Scale Graphs. *CoRR* abs/2010.05337 (2020). arXiv:2010.05337 https://arxiv.org/abs/2010.05337

[79] Jianlong Zhong and Bingsheng He. 2014. Medusa: Simplified Graph Processing on GPUs. *IEEE Trans. Parallel Distributed Syst.* 25, 6 (2014), 1543–1552. https://doi.org/10.1109/TPDS.2013.111

[80] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. 2019. AliGraph: A Comprehensive Graph Neural Network Platform. *Proc. VLDB Endow.* 12, 12 (2019), 2094–2105. https://doi.org/10.14778/3352063.3352127

[81] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A Computation-Centric Distributed Graph Processing System. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. USENIX Association, 301–316. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/zhu

Query Driven-Graph Neural Networks for Community Search: From Non-Attributed, Attributed, to Interactive Attributed

Yuli Jiang*[1], Yu Rong†[2], Hong Cheng*[3], Xin Huang‡[4], Kangfei Zhao†[5], Junzhou Huang†[6] *
*The Chinese University of Hong Kong, †Tencent AI Lab, ‡Hong Kong Baptist University, China
{[1]yljiang,[3]hcheng,[5]kfzhao}@se.cuhk.edu.hk, [2]yu.rong@hotmail.com, [4]xinhuang@comp.hkbu.edu.hk, [5]zkf1105@gmail.com, [6]jzhuang@uta.edu

ABSTRACT

Given one or more query vertices, Community Search (CS) aims to find densely intra-connected and loosely inter-connected structures containing query vertices. Attributed Community Search (ACS), a related problem, is more challenging since it finds communities with both cohesive structures and homogeneous vertex attributes. However, most methods for the CS task rely on inflexible pre-defined structures and studies for ACS treat each attribute independently. Moreover, the most popular ACS strategies decompose ACS into two separate sub-problems, i.e., the CS task and subsequent attribute filtering task. However, in real-world graphs, the community structure and the vertex attributes are closely correlated to each other. This correlation is vital for the ACS problem. In this vein, we argue that the separation strategy cannot fully capture the correlation between structure and attributes simultaneously and it would compromise the final performance.

In this paper, we propose Graph Neural Network (GNN) models for both CS and ACS problems, i.e., Query Driven-GNN (QD-GNN) and Attributed Query Driven-GNN (AQD-GNN). In QD-GNN, we combine the local query-dependent structure and global graph embedding. In order to extend QD-GNN to handle attributes, we model vertex attributes as a bipartite graph and capture the relation between attributes by constructing GNNs on this bipartite graph. With a Feature Fusion operator, AQD-GNN processes the structure and attribute simultaneously and predicts communities according to each attributed query. Experiments on real-world graphs with ground-truth communities demonstrate that the proposed models outperform existing CS and ACS algorithms in terms of both efficiency and effectiveness. More recently, an interactive setting for CS is proposed that allows users to adjust the predicted communities. We further verify our approaches under the interactive setting and extend to the attributed context. Our method achieves 2.37% and 6.29% improvements in F1-score than the state-of-the-art model without attributes and with attributes respectively.

PVLDB Reference Format:
Yuli Jiang, Yu Rong, Hong Cheng, Xin Huang, Kangfei Zhao, Junzhou Huang. Query Driven-Graph Neural Networks for Community Search: From Non-Attributed, Attributed, to Interactive Attributed. PVLDB, 15(6): 1243 - 1255, 2022.
doi:10.14778/3514061.3514070

*This work is done during Yuli's internship at Tencent AI Lab. Yu Rong and Hong Cheng are the corresponding authors.

Proceedings of the VLDB Endowment, Vol. 15, No. 6 ISSN 2150-8097.
doi:10.14778/3514061.3514070

1 INTRODUCTION

Graph is an essential data structure to represent entities and their relationships, e.g., social networks, protein-protein interaction networks, web graphs, and knowledge graphs, to name a few. Community, a subgraph of densely intra-connected and loosely inter-connected structure, naturally exists as a functional module in real-world graphs. Community Search (CS) [7, 11, 19, 21, 25, 37] is a vital application in graph analytics. Concretely, given any query vertices, CS aims to find a vertex set with cohesive structure according to the query, i.e., query-dependent communities. Attributed Community Search (ACS), a related but more challenging problem, has attracted a lot of attention recently [10, 11, 22, 23, 25]. Given any query vertex and attribute set, ACS aims at finding query-dependent communities with homogeneous attributes, which means the community members share similar attributes with the query attributes.

For the CS and the ACS problems, existing studies suffer from two serious limitations, that is, ***structure inflexibility*** and ***attribute irrelevance***. Structure inflexibility refers to the problem that most community search models are based on a pre-defined subgraph pattern, such as k-core [8, 10, 37], k-truss [1, 21, 22], k-clique [7, 44], and k-edge connected component (ECC) [6, 19]. The pre-defined subgraph pattern imposes a very rigid requirement on the topological structure of communities, which may not perfectly hold in real-world communities. Attribute irrelevance means existing models treat each attribute independently [10, 22]. However, in the real graphs, vertex attributes are not independent of each other. Ignoring such implicit relations would harm the quality of queried communities.

Figure 1 depicts a toy example illustrating the limitation of existing algorithms. The faculty hierarchy is a tree-like structure from the faculty dean, department chairman to the professors in each department. Using existing methods based on pre-defined subgraph patterns, we can only find a 1-core community of vertex 6 in H_1 and a 2-truss community in H_2, which are the entire graph. These k-core [37] and k-truss [21] patterns cannot discover the tree-like department communities owing to the structure inflexibility. For attributed community search, when querying the community of vertex 6 and attribute "ML", current methods [10, 22] find the community H_3 since they ignore the implicit relations between "ML", "DL" and "CV". Thus, existing studies suffer from these two inadequacies on structure and attribute respectively.

Moreover, for the ACS problem, existing studies [10, 22] usually adopt a two-stage strategy which first finds the candidate community by considering the topological structure only, and then performs a filtering on the candidate community by considering the attribute similarity. The two-stage strategy *treats the structure*

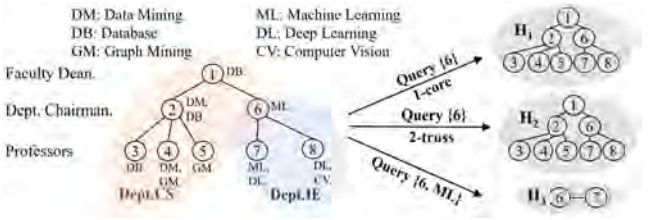


Figure 1: An attributed graph depicting a faculty hierarchy with two departments: Dept.CS and Dept.IE. Attributes represent research topics. For vertex 6, there is a ground-truth Dept.IE community (shown in blue) as vertices $6-8$ are close and work on similar topics. On the right, in response to queries on each arrow, there are three result communities found by existing algorithms, which are quite different from the ground-truth community.

cohesiveness and attribute homogeneity separately. But there is usually a correlation between structure and attribute, for instance, in protein-protein interaction networks, proteins with similar functions (i.e., attributes) are more likely to interact with each other [39]. Independently dealing with the structure and attribute would harm the quality of queried communities.

Inspired by the success of Graph Neural Network (GNN) [27] on combining attribute and structure in many graph problems, Gao *et al.* [14] proposed a GNN-based framework, ICS-GNN, to solve the community search problem in an interactive fashion (i.e., users can adjust predicted communities during the query process). Specifically, it enhances the non-attributed queries by the GNN model [27] which exploits the information from the existing vertex attributes in graphs. However, for every query, ICS-GNN re-trains the whole model. This re-training process is time-consuming and hinders its applications in real-world scenarios, especially for the online query case. On the other hand, even though ICS-GNN makes use of the attributes to enhance the community search performance, its model architecture cannot accept the query attributes as input. Therefore, ICS-GNN cannot be extended to support interactive attributed community search easily.

To address the above limitations, in this paper, we propose GNN-based models for both CS and ACS problems. For the CS problem, to address the structure inflexibility issue, we design a two-branch model: Query Driven-GNN (QD-GNN) to encode the information from both the query and graph. Concretely, QD-GNN contains two encoders, *Query Encoder* and *Graph Encoder*. *Query Encoder* encodes the structural information from query vertices and focuses on modeling the local topology around the queries. *Graph Encoder* combines the global structure and attributes to learn the query-independent node embeddings. As a learning-based model, QD-GNN can search communities without imposing any restriction on the community structure. Furthermore, we design an additional *Attribute Encoder* to extend QD-GNN to support the attributed community search. *Attribute Encoder* exploits a node-attribute bipartite graph to model the attribute relations and can encode more meaningful information from the attribute space. To process structure and attribute simultaneously, we employ a *Feature Fusion* component to fuse the information from different encoders and make the final output. Furthermore, we design a new query framework which detaches the model training from the online query stage. Therefore, our framework does not need the time-consuming re-training phase for online query applications.

To summarize, we make the following contributions.

- We propose a Query Driven-GNN model (QD-GNN) for community search, which combines the local query-dependent structure and global node embeddings. Given any query, QD-GNN only needs a model inference step and avoids the time-consuming re-training.
- To the best of our knowledge, this is the first work that proposes a GNN model for the attributed community search problem, called Attributed Query Driven-GNN (AQD-GNN). Our novel learning framework extends GNN into ACS through a node-attribute bipartite graph, and learns the community information from both the local structure and similar attributes of queries.
- We conduct extensive experiments on real-world data sets with ground-truth communities for performance evaluation. Experiments demonstrate that our model significantly outperforms state-of-the-art methods in terms of community quality with only 4.31 milliseconds average response time.
- We apply AQD-GNN to the interactive community search problem and extend it into the attributed context. Experiments show that our models can improve the performance of ICS-GNN [14] in both non-attributed and attributed manner with 2.37% and 6.29% improvements in F1-score respectively.

Roadmap. The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 gives some preliminaries. Section 4 presents the common framework of the proposed models. Section 5 introduces the QD-GNN model for community search problem, and Section 6 describes the AQD-GNN model for attributed community search. We present the experimental results in Section 7 and conclude the paper in Section 8.

2 RELATED WORK

Our study is closely related to community search (CS) and graph neural network (GNN).

Community Search. The problem of CS [37] is to find densely connected communities containing the query vertices. A comprehensive survey of CS models and existing approaches can be found in [11, 25]. Various community models have been proposed based on different cohesive graph patterns, including k-core [8, 37], k-truss [1, 21, 24], k-clique [7, 44], and k-edge connected component (ECC) [6, 19]. These pre-defined cohesive metrics are inflexible and can be too loose (e.g., k-core) or too tight (e.g., k-clique) to capture the topology structure of communities. If the real-world communities do not follow any of the above graph patterns, these models would fail to discover the true communities. A learning-based model ICS-GNN [14] has recently been proposed for interactive community search. ICS-GNN first finds a candidate subgraph starting from query vertices, then learns the node embeddings through applying GNN model on subgraph, and finally employs a BFS based algorithm to select the k-sized community with maximum GNN scores. ICS-GNN does not support attributed community search as the query only involves vertices but no attributes. It also needs to re-train the entire model for each query, which is costly for this online query problem.

For attributed community search, ACQ [10] and ATC [22] have been proposed, which aim to discover communities that contain query vertices and have similar attributes to the query attributes. ACQ is based on k-core and finds communities with the maximum number of common query attributes shared by community members. ATC finds k-truss communities with the maximum pre-defined attribute score. Both adopt a two-stage process. They first impose a pre-defined structural constraint to find candidate communities, then optimize functions of attribute score to select the most related communities. However, the attribute score functions ignore the similarities between attributes, and these two-stage methods fail to capture the correlation between structure and attribute. In this paper, we propose QD-GNN, which considers the cohesive structure and homogeneous attributes in an integrated way.

Graph Neural Network. Inspired by the huge success of neural networks in natural language processing and computer vision, many graph analytic problems have been solved via graph neural networks [27], such as node classification[4, 18, 35], graph classification [20, 29], drug discovery [31, 34, 43], adversarial attacks [2, 3, 5, 47] and graph algorithmic tasks [45, 46]. To build good models, the advanced techniques of pooling [13, 28, 32] and attention [12, 28, 40] have been developed. However, most learning models are designed for specific tasks based on graph embedding [30, 42] or end-to-end solutions [15, 36]. Existing GNN models cannot extend to attributed community search straightforwardly. To the best of our knowledge, we are the first to propose a GNN-based model for attributed community search and extend ICS-GNN to the attributed context as well.

3 PRELIMINARIES

In this section, we first introduce the notations and define the problems of CS and ACS formally, and then describe a general GNN as the foundation of our proposed models.

3.1 Definitions

Let $G(\mathcal{V}, \mathcal{E})$ be a graph with a set $\mathcal{V}$ of vertices and a set $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ of edges. Let $n = |\mathcal{V}|$ and $m = |\mathcal{E}|$ be the number of vertices and edges respectively. We denote $\mathcal{N}(\mathrm{v}) = \{\mathrm{u} \mid (\mathrm{u}, \mathrm{v}) \in \mathcal{E}\}$ as the neighborhood set of vertex v. Moreover, let $\mathcal{N}^+(\mathrm{v}) = \{\mathrm{v}\} \cup \mathcal{N}(\mathrm{v})$ be the vertex set containing v's neighbors and v itself.

Community Search (CS). For a graph $G(\mathcal{V}, \mathcal{E})$, given a vertex query set $\mathcal{V}_q \subseteq \mathcal{V}$, the problem of Community Search (CS) is to find the query-dependent community $C_q \subseteq \mathcal{V}$. Vertices in community C_q need to be densely intra-connected, i.e., having cohesive structure.

Let $G(\mathcal{V}, \mathcal{E}, \mathcal{F})$ be an attributed graph where $\mathcal{F} = \{\mathcal{F}_1, \ldots, \mathcal{F}_n\}$ is the set of vertex attributes and $\mathcal{F}_i$ is the attribute set of vertex v_i. Define $\hat{\mathcal{F}}$ as the union of all the vertex attribute sets, i.e., $\hat{\mathcal{F}} = \mathcal{F}_1 \cup \ldots \cup \mathcal{F}_n$. Let d be the number of unique attributes $d = |\hat{\mathcal{F}}|$. The attribute set of each vertex, e.g., $\mathcal{F}_i$, is encoded to a d-dimensional vector $\boldsymbol{f}_i$. For a keyword attribute f_k, if vertex v_i has this keyword, i.e., $\mathrm{f}_k \in \mathcal{F}_i$, then $\boldsymbol{f}_{ik} = 1$; otherwise, $\boldsymbol{f}_{ik} = 0$. For a numerical attribute f_j, $\boldsymbol{f}_{ij}$ is the value of vertex v_i on this attribute. Then the set of vertex attributes $\mathcal{F} = \{\mathcal{F}_1, \ldots, \mathcal{F}_n\}$ is encoded to an attribute matrix $\boldsymbol{F} = [\boldsymbol{f}_1, \ldots, \boldsymbol{f}_n]^T \in \mathbb{R}^{n \times d}$.

Attributed Community Search (ACS). For an attributed graph $G(\mathcal{V}, \mathcal{E}, \mathcal{F})$, given a query $\langle \mathcal{V}_q, \mathcal{F}_q \rangle$ where $\mathcal{V}_q \subseteq \mathcal{V}$ is a set of query vertices, and $\mathcal{F}_q \subseteq \hat{\mathcal{F}}$ is a set of query attributes, the problem of Attributed Community Search (ACS) is to find the query-dependent community $C_q \subseteq \mathcal{V}$. Vertices in community C_q need to be both structure cohesive and attribute homogeneous, i.e., vertices in a community are densely intra-connected in structure and attributes of these vertices are similar.

In this paper, we formulate the above two problems as a binary classification task. Given a query $q = \langle \mathcal{V}_q \rangle$ or $q = \langle \mathcal{V}_q, \mathcal{F}_q \rangle$, we classify the graph vertices into two classes (belonging to a community C_q of query q or not). We use the one-hot vector $\boldsymbol{c}_q \in \{0, 1\}^n$ to represent the output community C_q by a model $\mathcal{M}$. If the output value $\boldsymbol{c}_{q_k} = 1$, vertex v_k belongs to the result community C_q predicted by $\mathcal{M}$.

3.2 A General GNN Model

We introduce a general framework of Graph Neural Network (GNN) as the cornerstone of our models.

A GNN layer is known as a message passing procedure from neighborhoods. After the linear transformation of neighbors' hidden features, there are many alternative techniques within one layer, e.g., batch normalization technique [26]. We list one of the possible intra-layer processes in the layer-wise propagation function as:

$$\boldsymbol{h}_v^{(l+1)} = \mathrm{Dr}\left\{\phi\left(\mathrm{BN}[\mathrm{AGG}(\boldsymbol{h}_u^{(l)}\boldsymbol{W}^{(l+1)} + \boldsymbol{b}^{(l+1)}, \mathrm{u} \in \mathcal{N}^+(\mathrm{v}))]\right)\right\}, \tag{1}$$

where $\boldsymbol{h}_v^{(l+1)} \in \mathbb{R}^{d^{(l+1)}}$ is the learned new features of vertex v in the $(l+1)$-th layer, $\boldsymbol{h}_u^{(l)} \in \mathbb{R}^{d^{(l)}}$ is the hidden features of vertex u from the l-th layer, and the input feature $\boldsymbol{h}_v^{(0)} \in \mathbb{R}^d$ is the normalized form of attribute vector $\boldsymbol{f}_v$. $\boldsymbol{W}^{(l+1)} \in \mathbb{R}^{d^{(l)} \times d^{(l+1)}}$ and $\boldsymbol{b}^{(l+1)} \in \mathbb{R}^{d^{(l+1)}}$ are trainable weights. $\mathrm{AGG}(\cdot)$ is an aggregation function such as SUM, MAX, or MIN. $\mathrm{BN}(\cdot)$ is batch normalization [26] that reduces internal covariate shift. $\phi(\cdot)$ is the non-linear activation function, such as $\mathrm{ReLU}(\cdot)$. Last, $\mathrm{Dr}(\cdot)$ is the dropout method [38] to dilute the data and reduce the overfitting in neural networks.

For example, one of the most classical GNN models, Vanilla Graph Convolutional Network (Vanilla GCN) [27], is defined as:

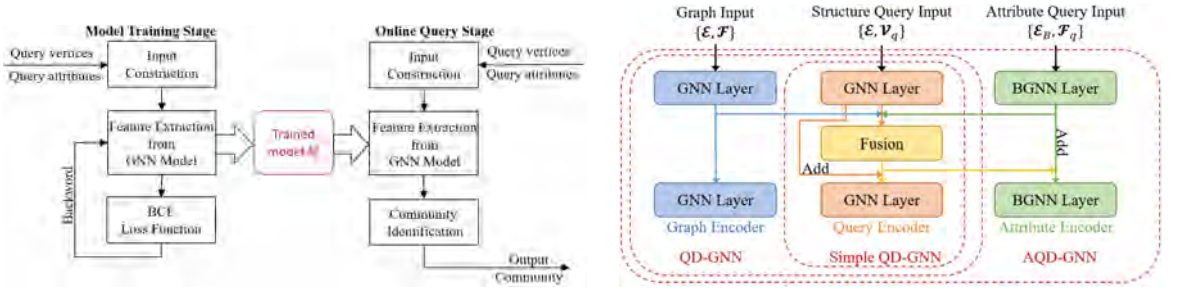
$$\boldsymbol{h}_v^{(l+1)} = \mathrm{Dr}\left\{\mathrm{ReLU}\left(\mathrm{SUM}(\{\frac{\boldsymbol{h}_u^{(l)}}{\sqrt{d'_\mathrm{u} d'_\mathrm{v}}}\boldsymbol{W}^{(l+1)} : \mathrm{u} \in \mathcal{N}^+(\mathrm{v})\})\right)\right\}, \tag{2}$$

which applies SUM as the aggregation operation, and $\mathrm{ReLU}(\cdot)$ as the activation function $\phi(\cdot)$ with the dropout method. In this GNN model, batch normalization is not adopted and Laplacian smoothing is employed where $d'_\mathrm{u} = d_\mathrm{u} + 1$ and d_u is the degree of vertex u.

In the following, we will focus on the way of aggregation in our proposed GNN models. The dropout, activation function, batch normalization and the trainable bias $\boldsymbol{b}$ described above are adopted in our models, and will be omitted in our following presentation.

4 THE QUERY FRAMEWORK

Before describing the detailed design of the proposed models, we introduce the common framework of our models for both CS and ACS problems. As Figure 2a shows, the proposed models consist of two main stages: *the model training stage* and *the online query stage*. Firstly, we train the embedding model $\mathcal{M}$ offline with the loss function in the model training stage as shown in Figure 2a (left). After that, in the online query stage, whenever the query comes, we apply the model from the training stage to predict the



(a) Framework of our proposed models. **(b) Intermediate design of model $\mathcal{M}$ with four components.**

Figure 2: The architecture of proposed models.

community without re-training, as Figure 2a (right) presents. This framework is highly flexible. In the following, we first introduce how to construct the inputs from the graph and queries in both stages. Then, we describe the two main stages respectively.

4.1 Input Construction

Since the GNN model $\mathcal{M}$ needs vectorized inputs, we introduce the vectorization scheme for the vertex set and attribute set.

Construct query vertices. We encode each query vertex set $\mathcal{V}_q \subseteq \mathcal{V}$ to a one-hot vector $\boldsymbol{v}_q \in \{0,1\}^n$. For a query $\mathcal{V}_q$, if vertex $v_i \subseteq \mathcal{V}_q$, $\boldsymbol{v}_{q_i} = 1$; otherwise, $\boldsymbol{v}_{q_i} = 0$. For example, when querying the community of vertex v_6 in Figure 1, the encoded vector is $\boldsymbol{v}_q = [0,0,0,0,0,1,0,0]^T$.

Construct query attributes. Similar to query vertices, we encode each query attribute set $\mathcal{F}_q \subseteq \hat{\mathcal{F}}$ to a one-hot vector $\boldsymbol{f}_q \in \{0,1\}^d$, where $d = |\hat{\mathcal{F}}|$ is the number of unique attributes.

The encoded query vertex set and query attribute set are then submitted to our proposed GNN models as input features.

4.2 Model Training Stage

In the model training stage, with a set of training queries as input, we iteratively train the embedding model $\mathcal{M}$ offline through the Binary Cross Entropy (BCE) loss function and obtain a trained model for the online query stage.

Given a set of training queries $Q_{\text{train}} = \{q_1, q_2, ...\}$ and corresponding ground-truth communities $C_{\text{train}} = \{C_{GT1}, C_{GT2}, ...\}$, we train a GNN model $\mathcal{M}$ to minimize the loss function to fit the training data. Given a validation query set Q_{val} and corresponding ground-truth communities C_{val}, we select the parameters of model $\mathcal{M}$ and threshold $\gamma \in [0,1]$ which achieve the best performance in the validation set. The queries in Q_{train} and Q_{val} can be attributed $q = \{\mathcal{V}_q, \mathcal{F}_q\}$ for ACS or non-attributed $q = \{\mathcal{V}_q\}$ for CS.

First, we construct all query inputs as one-hot vectors. Then we repeatedly input queries into the model $\mathcal{M}$, i.e., Simple QD-GNN, QD-GNN or AQD-GNN, which will be introduced in Section 5 and Section 6. With the model $\mathcal{M}$'s output $\boldsymbol{h}_q$ for each query q in an iteration, we compute BCE loss function and gradients of the model parameters. The gradients are propagated backward to update $\mathcal{M}$ at the end of this iteration. With the updated parameters, $\mathcal{M}$ moves to the next iteration, outputs $\boldsymbol{h}_q$, calculates loss and back propagates gradients until convergence. The loss function of the three proposed models is the same and we describe it formally in the following.

Loss Function. We formulate community search as a binary classification problem. Assume that $\boldsymbol{h}_q \in \mathbb{R}^n$ is the output of $\mathcal{M}$ for query q after the Sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$, where $\boldsymbol{h}_{q_v} \in [0,1]$ represents the output for vertex v. $\boldsymbol{y}_q \in \{0,1\}^n$ represents the ground-truth vector for query q. $\boldsymbol{y}_{q_v} = 1$ if and only if vertex $v \in C_{GTq}$; otherwise, $\boldsymbol{y}_{q_v} = 0$. Then we utilize Binary Cross Entropy (BCE) function as the loss function to minimize the BCE between the model output $\boldsymbol{h}_q$ and the ground-truth label $\boldsymbol{y}_q$ for q. The optimization loss function can be formulated as:

$$\min \mathcal{L} = \sum_{q \in Q_{\text{train}}} \frac{1}{n} \sum_{i=1}^{n} -(\boldsymbol{y}_{q_i} \log(\boldsymbol{h}_{q_i}) + (1 - \boldsymbol{y}_{q_i}) \log(1 - \boldsymbol{h}_{q_i})). \tag{3}$$

Algorithm 1 Constrained BFS for Community Identification

Input: Graph: $G = (\mathcal{V}, \mathcal{E})$, a query vertex set: $\mathcal{V}_q$, a model output vector: $\boldsymbol{h}_q$, a threshold: γ.
Output: a vertex set of community : C_q.

1: Initialize set $Q = \mathcal{V}_q$, $C_q = \mathcal{V}_q$
2: **while** Q is not empty **do**
3: select a vertex v from Q
4: **for** u $\in \mathcal{N}$(v) and $\boldsymbol{h}_{q_u} \geq \gamma$ **do**
5: $Q \leftarrow Q \cup \{u\}$
6: $C_q \leftarrow C_q \cup \{u\}$
7: **return** C_q;

4.3 Online Query Stage

In the online query stage, we utilize the well-trained model $\mathcal{M}$ and threshold γ from the model training stage to process the online query q and produce the community C_q without re-training. We first construct query inputs as one-hot vectors. Then the constructed vectors are fed into model $\mathcal{M}$, which only runs once and outputs the vector $\boldsymbol{h}_q$. To ensure the connectivity between query vertices and community members, we employ a constrained Breadth-First Search (BFS) starting from the query vertices in Algorithm 1. When visiting vertex u, if $\boldsymbol{h}_{q_u} \geq \gamma$ (line 4), we add vertex u to the output community C_q (line 6).

Please note that the connectivity of the output community also depends on the user-specified query vertices. If the induced subgraph of the query vertices is connected, then our models are guaranteed to find a connected community. If the induced subgraph of the query vertices is not connected, our models may still find a connected community through some bridging vertices. But there is possibility that the discovered community is not connected as one component, especially when the query vertices are distant or disconnected in the graph. In this case, our models can still find some connected components, each of which contains part of the query vertices, as the answer community.

Figure 3: Query propagation paths in Query Encoder.

5 QD-GNN MODEL FOR CS

In this section, we introduce the construction of the embedding model $\mathcal{M}$ in the proposed framework for community search. We first propose a Simple task-oriented Query Driven-Graph Neural Network (Simple QD-GNN) and then design useful functional encoders to improve it as QD-GNN model. As Figure 2a shows, with query vectors as input, the Simple QD-GNN or QD-GNN model $\mathcal{M}$ outputs $\boldsymbol{h}_q$ into the BCE loss function during the training process. In the online query stage, the model output $\boldsymbol{h}_q$ is translated into community members as described in Section 4.

5.1 Simple QD-GNN

The Simple QD-GNN model is designed based on the general GNN introduced in Section 3.2 and uses query vector $\boldsymbol{v}_q$ as the input features of the model. This model input enables query-centered structural propagation, i.e., propagating from the query vertices to its neighborhood, to better capture the local query structure information.

We name this query driven propagation as *Graph Encoder*. In order to fully make use of vertex features in each layer, Query Encoder is designed to equip with a self feature modeling [12]. The inter-layer propagation function for vertex v is formally defined as:

$$\boldsymbol{h}_{Q_v}^{(l+1)} = \boldsymbol{h}_{Q_v}^{(l)} \boldsymbol{W}_{Q_{\text{self}}}^{(l+1)} + \text{SUM}(\{\boldsymbol{h}_{Q_u}^{(l)} \boldsymbol{W}_Q^{(l+1)} : \text{u} \in \mathcal{N}^+(\text{v})\}), \quad (4)$$

where the first component emphasizes the self features (hidden features of the vertex v) with learnable weight parameter matrices $W_{Q_{\text{self}}}^{(l+1)} \in \mathbb{R}^{d^{(l)} \times d^{(l+1)}}$. The second component is similar to Eq. (1) with a subscript Q, and chooses SUM as the aggregation function as Vanilla GCN [27] does. Similarly, $\boldsymbol{W}_Q^{(l+1)} \in \mathbb{R}^{d^{(l)} \times d^{(l+1)}}$ is the trainable weight matrix , $\boldsymbol{h}_{Q_v}^{(l+1)} \in \mathbb{R}^{d^{(l+1)}}$ is the learned new features of vertex v in the $(l+1)$-th layer of Query Encoder. Different from Eq. (1), the input feature of the first layer $\boldsymbol{h}_{Q_v}^{(0)}$ is the one-hot query vector $\boldsymbol{v}_{q_v}$.

Example. *We follow the example in Figure 1 and show the propagation paths in Figure 3. For query $\mathcal{V}_q = \{\text{v}_8\}$ highlighted in Figure 3a, the query vector $\boldsymbol{v}_q$ is $[0, 0, 0, 0, 0, 0, 0, 1]^T$. According to Eq.* (4)*, in the first layer, the query information propagates to the neighbor of* v_8*, i.e.,* v_6 *as depicted in Figure 3b. Then, the 2-hop neighbors of the query vertex,* v_1 *and* v_7*, acquire the knowledge from* v_6 *in the second layer as depicted in Figure 3c.*

5.2 QD-GNN

Recent studies [9, 14, 41] have found that attributes on graph vertices can be leveraged for structural learning problems, for example, link prediction [9] and community search [14]. Inspired by their findings, we design an improved QD-GNN model based on Simple QD-GNN and Vanilla GCN [27]. Similar to ICS-GNN [14], QD-GNN combines the network structure and vertex attributes to solve the community search problem.

5.2.1 Overview. Figure 2b presents the architecture overview of QD-GNN model, which consists of two convolution branches (*Graph Encoder* and *Query Encoder*) and a *Feature Fusion* operator. *Graph Encoder* provides the query-independent information with both graph structure and vertex attributes as input, i.e., the edge set $\mathcal{E}$ and vertex attribute set $\mathcal{F}$. *Query Encoder* (the same as that in Simple QD-GNN) provides the interface for query vertices and learns the query-specific local topology features. It takes the input of graph structure and query vertices, i.e., the edge set $\mathcal{E}$ and query vertices $\mathcal{V}_q$. The *Feature Fusion* operator combines the above encoder embedding results and obtains the final query-specific output vectors. This fusion makes use of both global graph knowledge and local query information which can achieve a good balance, and finally obtains the model output $\boldsymbol{h}_q$ for each query q.

5.2.2 Graph Encoder. Graph Encoder focuses on global graph structure and vertex attributes, both of which are independent of queries. We apply the layer-wise forward propagation of the general GNN to construct Graph Encoder, which has been introduced in Section 3.2. Similar to Simple QD-GNN, the forward layer of Graph Encoder is defined with a self feature modeling [12] as:

$$\boldsymbol{h}_{G_v}^{(l+1)} = \boldsymbol{h}_{G_v}^{(l)} \boldsymbol{W}_{G_{\text{self}}}^{(l+1)} + \text{SUM}(\{\boldsymbol{h}_{G_u}^{(l)} \boldsymbol{W}_G^{(l+1)} : \text{u} \in \mathcal{N}^+(\text{v})\}), \quad (5)$$

where the notations are the same as Eq. (1) with a subscript G, and $W_{G_{\text{self}}}^{(l+1)} \in \mathbb{R}^{d^{(l)} \times d^{(l+1)}}$ are the weight parameter matrices. The input feature of vertex v in the first layer $\boldsymbol{h}_{G_v}^{(0)} \in \mathbb{R}^d$ is the normalized attribute vector $\boldsymbol{f}_v$ encoded in Section 3.1. Graph Encoder propagates the attribute information through graph structure and learns query-independent knowledge.

Example. *We follow the example in Figure 1 to illustrate how Graph Encoder works. For vertex* v_8*, in the first layer, its attributes ("DL" and "CV") are propagated to its neighbor, vertex* v_6*, with a learnable weight. At the same time, the attribute of vertex* v_6 *("ML") is also propagated to vertex* v_8*. In the next layer, those attributes are propagated to their neighbors respectively as well. By this propagation, the attributes of vertices* v_6*,* v_7 *and* v_8 *become more similar. This information is used by the Feature Fusion operator to identify the community members more accurately.*

5.2.3 Query Encoder. Query Encoder is the same as that of Simple QD-GNN and provides an interface for query vertices and obtains the local structure knowledge. Inputs of the Query Encoder are based on the graph topology (graph edges $\mathcal{E}$) and structural query (query vertices $\mathcal{V}_q$). The inter-layer propagation function of Query Encoder is the same as that in Eq. (4).

5.2.4 Feature Fusion. The Feature Fusion operator combines output features learned by the above two encoders, and balances the global and local information to get the final output of QD-GNN. The inputs of Feature Fusion are based on the output of the two encoders, i.e., $\boldsymbol{h}_G$ and $\boldsymbol{h}_Q$. It fuses them and transmits the fusion result to Query Encoder as shown in Figure 2b.

Based on the output of the two encoders, the forward layer of Feature Fusion is formulated as:

$$\boldsymbol{h}_{FF_v}^{(l+1)} = \text{AGG}(\boldsymbol{h}_{G_v}^{(l+1)}, \boldsymbol{h}_{Q_v}^{(l+1)}), \quad (6)$$

where $\boldsymbol{h}_{FF_v}^{(l+1)}$ is the output of Feature Fusion for vertex v and also the final output of the entire QD-GNN model in the $(l+1)$-th layer, $\text{AGG}(\cdot)$ is the aggregation function (e.g., Concatenation, SUM, etc.),

Algorithm 2 The k-Layer QD-GNN Propagation

Input: Graph: $G = (\mathcal{V}, \mathcal{E}, \mathcal{F})$,
a set of queries: $Q = \{\mathcal{V}_{q_1}, \mathcal{V}_{q_2}, \dots\}$,
QD-GNN model: $\mathcal{M} = \{\boldsymbol{h}_Q, \boldsymbol{h}_G, \boldsymbol{h}_{FF}\}$.
Output: a set of output vectors: $\mathcal{H} = \{\boldsymbol{h}_{q_1}, \boldsymbol{h}_{q_2}, \dots\}$.

1: Construct attribute matrix $\boldsymbol{F}$ for $\mathcal{F}$
2: $\mathcal{H} \leftarrow \varnothing$
3: **for** each $\mathcal{V}_q \in Q$ **do**
4: Construct one-hot vector $\boldsymbol{v}_q$ for $\mathcal{V}_q$, initialize $\boldsymbol{h}_Q^{(0)}$ with $\boldsymbol{v}_q$
5: Initialize $\boldsymbol{h}_G^{(0)}$ with $\boldsymbol{F}$
6: $\boldsymbol{h}_Q^{(1)} \leftarrow \text{Propg}(\boldsymbol{h}_Q^{(0)}, \mathcal{E})$ in Eq. (4)
7: $\boldsymbol{h}_G^{(1)} \leftarrow \text{Propg}(\boldsymbol{h}_G^{(0)}, \mathcal{E})$ in Eq. (5)
8: $\boldsymbol{h}_{FF}^{(1)} \leftarrow \text{AGG}(\boldsymbol{h}_G^{(1)}, \boldsymbol{h}_Q^{(1)})$ in Eq. (6)
9: $l \leftarrow 1$
10: **while**$(l < k)$ **do**
11: $\boldsymbol{h}_Q^{(l+1)} \leftarrow \text{Propg}(\boldsymbol{h}_{FF}^{(l)}, \mathcal{E})$ in Eq. (8)
12: $\boldsymbol{h}_G^{(l+1)} \leftarrow \text{Propg}(\boldsymbol{h}_G^{(l)}, \mathcal{E})$ in Eq. (5)
13: $\boldsymbol{h}_{FF}^{(l+1)} \leftarrow \text{AGG}(\boldsymbol{h}_G^{(l+1)}, \boldsymbol{h}_Q^{(l+1)})$ in Eq. (6)
14: $l \leftarrow l + 1$
15: $\mathcal{H} \leftarrow \mathcal{H} \cup \boldsymbol{h}_{FF}^{(k)}$
16: **return** $\mathcal{H}$;

and $\boldsymbol{h}_{G_v}^{(l+1)}$, $\boldsymbol{h}_{Q_v}^{(l+1)}$ are the outputs of each encoder for vertex v in the $(l+1)$-th layer respectively.

For Graph Encoder, we do not use the fusion result and just use the output of Graph Encoder itself in the l-th layer $\boldsymbol{h}_G^{(l)}$ as the input of the $(l+1)$-th layer. Thus, we keep Graph Encoder independent of query information in the intermediate layer. This query-independent features provide stable "prior" knowledge about the graph and supply additional information for community search problem, which makes QD-GNN a stronger model.

For Query Encoder, we replace the feature propagation between neighbors as the fusion features in the intermediate layers. This fusion operation transmits the vertex attributes and global structure features into Query Encoder and delivers these features around query vertices. We define $\hat{\boldsymbol{h}_Q}$ as the input feature of each layer which can be formally written as:

$$\hat{\boldsymbol{h}}_{Q_i}^{(l)} = \begin{cases} \boldsymbol{v}_{q_i}, & \text{if } l = 0; \\ \boldsymbol{h}_{FF_i}^{(l)}, & \text{otherwise.} \end{cases} \tag{7}$$

The propagation function of Query Encoder can be rewritten as:

$$\boldsymbol{h}_{Q_v}^{(l+1)} = \boldsymbol{h}_{Q_v}^{(l)} \boldsymbol{W}_{Q_{\text{self}}}^{(l+1)} + \text{SUM}(\{\hat{\boldsymbol{h}}_{Q_u}^{(l)} \boldsymbol{W}_Q^{(l+1)} : \text{u} \in \mathcal{N}^+(\text{v})\}). \tag{8}$$

5.2.5 Algorithm. The QD-GNN model for the community search problem is presented in Algorithm 2. For easy description, we simplify the propagation function in each encoder as $\text{Propg}(\boldsymbol{h}, \mathcal{E})$, which means propagating feature $\boldsymbol{h}$ through edges in $\mathcal{E}$. At the beginning, we construct the feature matrix for the graph (line 1) and set the output as empty (line 2). For each query, we also construct the query vector and initialize Query Encoder and Graph Encoder (line 4-5). In the first layer (line 6-9), Query Encoder and Graph Encoder propagate their input features through the graph edges (line 6-7), and Feature Fusion fuses the output of them (line 8). In the intermediate layers (line 10-14), Query Encoder utilizes the fused feature from Feature Fusion (line 11), while Graph Encoder takes its own output $\boldsymbol{h}_G$ as the input feature to remain independent of the query (line 12). The final output of QD-GNN is the fused feature $\boldsymbol{h}_{FF}$ and we add it into the output set $\mathcal{H}$ (line 15).

6 AQD-GNN MODEL FOR ACS

In this section, we extend QD-GNN by incorporating the query attributes and propose the GNN model for attributed community search, named Attributed Query Driven-Graph Neural Network (AQD-GNN). We first identify the challenges of attributed community search when using GNN models. Then, we describe the components of AQD-GNN one by one in detail.

6.1 Challenges

Different from community search [1, 8, 14, 21, 37], the attributed community search task [10, 22] needs to integrate the query attributes into models. However, the meaning of query attributes F_q and the dimension of query attributes vector f_q are different from those of query vertices. It is not feasible to input query attribute information as we handle query vertices in Section 5.

The ICS-GNN model [14] utilizes the query vertex information as the labels of vertices, and aligns the output embedding and the labels through a loss function. Since previous studies always focus on tasks at the level of vertices and edges, such as node classification and link prediction, but not at the attribute level for attributed queries, the design of their loss functions also centers on the vertices. The BCE loss function in Eq. (3) is an example, which focuses on the class of each vertex. Therefore, ICS-GNN cannot incorporate the query attributes in the loss function directly and thus is not able to extend to the ACS problem.

The similar phenomenon can be observed from the QD-GNN model, which considers query vertices as the input features and propagates the query information via edges to find local structures surrounding the query vertices. But the query attributes cannot be easily incorporated as model input due to the different dimensionality. Even if we have a mechanism to take query attributes as input features, this attribute information can only propagate to adjacent vertices via graph topology by QD-GNN, but cannot reach vertices having similar attributes to the query attributes, as ACS aims to do.

The above discussions reveal that *incorporating query attributes into the learning model* and *identifying the vertices with similar attributes automatically* are two key issues to be addressed in applying GNN models into the ACS problem. In AQD-GNN, we design a bipartite graph to represent the relations between vertices and attributes. Leveraging this bipartite graph, AQD-GNN can accept an input of query attributes and translate this query attribute knowledge into vertex knowledge. Finally, AQD-GNN can find the vertices which have similar attributes with the query attributes.

6.2 Overview

AQD-GNN takes an attributed graph $G = (\mathcal{V}, \mathcal{E}, \mathcal{F})$ and a group of attributed queries $q = \langle v_q, f_q \rangle$ vectorized as inputs, and predicts the community vector $\boldsymbol{h}_q$ as outputs for each query q. Figure 2b illustrates the inter-layer design of AQD-GNN, which consists of a Feature Fusion operator and three GNN components: Graph Encoder, Query Encoder and Attribute Encoder. Note that Graph Encoder and Query Encoder are the same as those of QD-GNN in Section 5.2. Attribute Encoder is a new component specifically designed for ACS. Accordingly, Feature Fusion needs to be revised due to the new Attribute Encoder. In the following, we describe Attribute Encoder and the revised Feature Fusion operator.

Attribute Encoder. Attribute Encoder serves as the interface of query attributes and provides attribute information related to

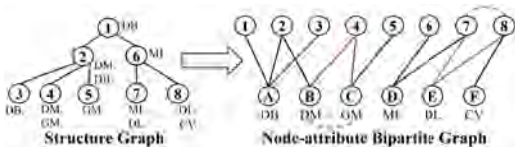


Figure 4: An example of node-attribute bipartite graph.

queries. It views each attribute as an individual vertex and models vertex attributes as a bipartite graph between vertex set $\mathcal{V}$ and attribute set $\hat{\mathcal{F}}$. With this bipartite graph, query attributes can be inputted into the attribute side directly and propagated between the vertex side and attribute side. Through this propagation, Attribute Encoder learns query-specific attribute node embeddings and identifies the vertices with attributes similar to queries.

Feature Fusion. The Feature Fusion component combines all the above embeddings and obtains the final query-specific output of the ACS problem. It takes the outputs of the three encoders as inputs, mixes global graph features and local query features, fuses structure and attribute information, and balances them to get an accurate community. Note that the final output of the entire model is the fused result in the last layer.

In the following sections, we will illustrate the detailed working mechanism of Attribute Encoder and Feature Fusion.

6.3 Attribute Encoder

The Attribute Encoder provides the interface for query attributes $\mathcal{F}_q$ and produces the vertex embeddings based on the related attributes of queries. Attribute Encoder aims to figure out the underlying relationship among different attributes and find the related attribute of queries. In addition, as analyzed in Section 6.1, Attribute Encoder needs to represent such attribute information in the form of vertices since the final output community is represented by a set of vertices.

To achieve the above goals, we model a bipartite graph called node-attribute bipartite graph $BG(\mathcal{V}, \hat{\mathcal{F}}, \mathcal{E}_B)$. For clarity, we call the vertices in the structure graph as nodes here. This bipartite graph is formed by two vertex sets: graph nodes $\mathcal{V}$ and graph attributes $\hat{\mathcal{F}}$. An edge between node v_i and attribute f_j is added to the edge set $\mathcal{E}_B$, if and only if node v_i has attribute f_j, i.e., $f_j \in \mathcal{F}_i$.

Example. *Figure 4 illustrates the node-attribute bipartite graph for the example in Figure 1. Based on the structure graph with node attributes on the left, we construct a node-attribute bipartite graph shown in Figure 4 (right), where the node set $\mathcal{V} = \{1, \ldots, 8\}$ is on the top and the attribute set $\hat{\mathcal{F}} = \{A, \ldots, F\}$ is at the bottom. Since node 4 has two attributes "DM" and "GM" in the structure graph, node 4 is adjacent to attribute B ("DM") and attribute C ("GM") as connected by red lines in Figure 4 (right).*

We apply Bipartite Graph Neural Network (BGNN) [17] on the constructed bipartite graph. BGNN consists of propagations in two directions between two vertex sets. In our node-attribute bipartite graph, the propagations are from the attribute side to the node side (denoted as A→N), and also from the node side to the attribute side (denoted as N→A).

Propagation A→N. We encode each query attribute set $\mathcal{F}_q \subseteq \hat{\mathcal{F}}$ to a one-hot vector $\boldsymbol{f}_q \in \{0,1\}^d$ as described in Section 4.1, where $d = |\hat{\mathcal{F}}|$. Benefiting from the node-attribute bipartite graph, we are able to take the query attribute vector $\boldsymbol{f}_q$ as input features in the attribute side, and propagate this attribute information from the attribute side to the node side.

Example. *When the attribute query is $\mathcal{F}_q = \{$"DL"$\}$, the one-hot vector is $\boldsymbol{f}_q = [0,0,0,0,1,0]^T$ according to the order of attribute vertices A to F in Figure 4. The query attribute information of "DL" will propagate to node 7 and node 8, the neighbors of "DL" vertex, through the blue edges in the bipartite graph of Figure 4.*

This propagation from the attribute side to the node side (A→N) collects attribute features for each node and translates the attribute features to node features. The layer-wise propagation function of A→N in BGNN is formally defined as:

$$\boldsymbol{h}_{N_u}^{(l+1)} = \mathrm{SUM}(\{\boldsymbol{h}_{A_\mathrm{f}}^{(l)} \boldsymbol{W}_{A\to N}^{(l+1)}, \mathrm{f} \in \mathcal{N}_B(\mathrm{u})\}), \tag{9}$$

where node $u \in \mathcal{V}$, attribute $\mathrm{f} \in \hat{\mathcal{F}}$, and $\mathcal{N}_B(\mathrm{u})$ is the neighbor set of node u in the bipartite graph. $\boldsymbol{h}_{N_u}^{(l+1)} \in \mathbb{R}^{d^{(l+1)}}$ is the hidden feature of node u in the $(l+1)$-th layer, $\boldsymbol{h}_{A_\mathrm{f}}^{(l)} \in \mathbb{R}^{d^{(l)}}$ is the input feature of attribute f in the l-th layer, and $\boldsymbol{W}_{A\to N}^{(l+1)} \in \mathbb{R}^{d^{(l)} \times d^{(l+1)}}$ is a learnable parameter matrix in propagation from the attribute side to the node side. The input feature of attribute f in the first layer is equal to the value of attribute f in the one-hot query attribute vector, i.e., $\boldsymbol{h}_{A_\mathrm{f}}^{(0)} = \boldsymbol{f}_{q\mathrm{f}}$.

Propagation N→A. After the propagation from the attribute side to the node side in the $(l+1)$-th layer, the learned features also need to be transmitted back to form an iterative propagation in the bipartite graph. Here, we also emphasize the attribute in the last layer and add a self feature modeling [12]. Similarly, the layer-wise propagation function from the node side to the attribute side (N→A) in BGNN is defined as:

$$\boldsymbol{h}_{A_\mathrm{f}}^{(l+1)} = \boldsymbol{h}_{A_\mathrm{f}}^{(l)} \boldsymbol{W}_{\mathrm{self}}^{(l+1)} + \mathrm{SUM}(\{\boldsymbol{h}_{N_u}^{(l+1)} \boldsymbol{W}_{N\to A}^{(l+1)} : \mathrm{u} \in \mathcal{N}_B(f)\}), \tag{10}$$

where the notations are the same as Eq. (9). $\boldsymbol{h}_{N_u}^{(l+1)}$ is the input features of node u in propagation N→A, which is learned in Eq. (9). $\mathcal{N}_B(\mathrm{f})$ is the neighbor set of attribute f in the bipartite graph. $\boldsymbol{W}_{N\to A}^{(l+1)} \in \mathbb{R}^{d^{(l)} \times d^{(l+1)}}$ is a learnable parameter matrix in the propagation from the node side to the attribute side, and $\boldsymbol{W}_{\mathrm{self}}^{(l+1)} \in \mathbb{R}^{d^{(l)} \times d^{(l+1)}}$ is the self feature parameter matrix in the $(l+1)$-th layer.

With these two propagations, Attribute Encoder can employ the query attribute as input features and transmit this attribute information through the node-attribute bipartite graph. Propagation A→N transforms the attribute features $\boldsymbol{h}_A$ into node features $\boldsymbol{h}_N$. Propagation N→A translates the node features $\boldsymbol{h}_N$ back to attribute features $\boldsymbol{h}_A$ and provides the input of propagation A→N in the next layer. With these bidirectional propagations, the features can spread in the bipartite graph and BGNN can be superimposed to multiple layers. Note that the node features $\boldsymbol{h}_N$ are the output of Attribute Encoder to Feature Fusion, since the community search problem focuses on the node and other encoders also provide node embeddings rather than attribute embeddings.

6.4 Feature Fusion

The Feature Fusion operator combines the output features of the three encoders, balances the global graph and local query knowledge, and mixes the structure and attribute information to obtain the final output of the AQD-GNN model.

The forward layer of Feature Fusion is formulated as:

$$\boldsymbol{h}_{FF_v}^{(l+1)} = \text{AGG}(\boldsymbol{h}_{G_v}^{(l+1)}, \boldsymbol{h}_{Q_v}^{(l+1)}, \boldsymbol{h}_{N_v}^{(l+1)}), \tag{11}$$

where $\boldsymbol{h}_{FF_v}^{(l+1)}$ is the fused feature of node v and also the final output of the AQD-GNN model in the $(l+1)$-th layer, $\text{AGG}(\cdot)$ is the aggregation function (e.g., Concatenation, SUM, etc.), and $\boldsymbol{h}_{G_v}^{(l+1)}$, $\boldsymbol{h}_{Q_v}^{(l+1)}$, $\boldsymbol{h}_{N_v}^{(l+1)}$ are the outputs of the three encoders in the $(l+1)$-th layer respectively. Note that $\boldsymbol{h}_{N_v}^{(l+1)}$ is the hidden features of the node side in Attribute Encoder.

In Eq. (11), we aggregate the three encoders to fuse all types of node embeddings. In order to consider the correlation between structure and attribute and process these two types of information simultaneously, we replace the input node features in the intermediate layers with the fused feature $\boldsymbol{h}_{FF}$ in Query Encoder and Attribute Encoder as shown in Figure 2b. Graph Encoder just uses the output of itself in the l-th layer $\boldsymbol{h}_G^{(l)}$ as the input of the $(l+1)$-th layer to capture the global query-independent node embeddings, as Feature Fusion does in QD-GNN. For Query Encoder, this fusion operation transmits the query-specific attribute features and global graph features into Query Encoder and delivers these features between vertices. Similar to Feature Fusion in QD-GNN, we employ $\hat{\boldsymbol{h}}_Q$ in Eq. (7) as the input features for Query Encoder, and rewrite the propagation function in Eq. (8). For Attribute Encoder, Feature Fusion enriches the features passed on the bipartite graph with local query structure and global graph features. Similar to Query Encoder, we replace the input node features in Eq. (10) with fused features when propagating from the node side to the attribute side. We define $\hat{\boldsymbol{h}}_N$ as the input node features:

$$\hat{\boldsymbol{h}}_{N_u}^{(l)} = \boldsymbol{h}_{FF_u}^{(l)}. \tag{12}$$

In this way, the structure features and attribute features learned by AQD-GNN can influence each other and these two encoders are correlated. Thus AQD-GNN is able to learn local structure and related attribute information of queries simultaneously. AQD-GNN provides an end-to-end attributed community search model, which takes queries as input and produces community vectors as answers.

6.5 Algorithm

Algorithm 3 describes the k-layer propagation of AQD-GNN. AQD-GNN first constructs the attribute matrix from $\mathcal{F}$, and builds an empty output set $\mathcal{H}$ (line 1-2). For each query, AQD-GNN constructs the one-hot vectors for both query vertex set and query attribute set, and initializes three encoders with them (line 3-6). In the first layer (line 7-11), Query Encoder propagates query vertices in the structure graph (line 7), Graph Encoder propagates vertex attributes in the graph (line 8), and Attribute Encoder propagates the query attributes from the attribute side to the node side in the bipartite graph (line 9). Feature Fusion fuses the output features of the three encoders (line 10). In the intermediate layers (line 12-18), Query Encoder propagates the fused features in graph (line 13), Graph Encoder still propagates the query-independent features from itself $\boldsymbol{h}_G$ (line 14), and Attribute Encoder utilizes the fused features $\boldsymbol{h}_{FF}$ as node side features and transmits node features back to attribute features (line 15). Then, Attribute Encoder is able to acquire the node hidden features in the next layer through propagating attribute features to the node side in the bipartite graph (line 16). Feature Fusion fuses the three encoders (line 17). The final output of AQD-GNN is the fused result in the last layer, which is added into the output set $\mathcal{H}$ (line 19).

Algorithm 3 The k-Layer AQD-GNN Propagation

Input: Graph: $G = (\mathcal{V}, \mathcal{E}, \mathcal{F})$,
a set of attributed queries: $\mathcal{Q} = \{\{\mathcal{V}_{q_1}, \mathcal{F}_{q_1}\}, \{\mathcal{V}_{q_2}, \mathcal{F}_{q_2}\}, \ldots, \}$,
AQD-GNN model: $\mathcal{M} = \{\boldsymbol{h}_Q, \boldsymbol{h}_G, \{\boldsymbol{h}_N, \boldsymbol{h}_A\}, \boldsymbol{h}_{FF}\}$.
Output: a set of output vectors: $\mathcal{H} = \{\boldsymbol{h}_{q_1}, \boldsymbol{h}_{q_2} \ldots, \}$.

1: Construct attribute matrix $\boldsymbol{F}$ for $\mathcal{F}$
2: $\mathcal{H} \leftarrow \varnothing$
3: **for** each $\{\mathcal{V}_q, \mathcal{F}_q\} \in \mathcal{Q}$ **do**
4: Construct one-hot vector $\boldsymbol{v}_q$ for $\mathcal{V}_q$, initialize $\boldsymbol{h}_Q^{(0)}$ with $\boldsymbol{v}_q$
5: Initialize $\boldsymbol{h}_G^{(0)}$ with $\boldsymbol{F}$
6: Construct one-hot vector $\boldsymbol{f}_q$ for $\mathcal{F}_q$, Initialize $\boldsymbol{h}_A^{(0)}$ with $\boldsymbol{f}_q$
7: $\boldsymbol{h}_Q^{(1)} \leftarrow \text{Propg}(\boldsymbol{h}_Q^{(0)}, \mathcal{E})$ in Eq. (4)
8: $\boldsymbol{h}_G^{(1)} \leftarrow \text{Propg}(\boldsymbol{h}_G^{(0)}, \mathcal{E})$ in Eq. (5)
9: $\boldsymbol{h}_N^{(1)} \leftarrow \text{Propg}(\boldsymbol{h}_A^{(0)}, \mathcal{E}_B)$ in Eq. (9)
10: $\boldsymbol{h}_{FF}^{(1)} \leftarrow \text{AGG}(\boldsymbol{h}_G^{(1)}, \boldsymbol{h}_Q^{(1)}, \boldsymbol{h}_N^{(1)})$ in Eq. (11)
11: $l \leftarrow 1$
12: **while**$(l < k)$ **do**
13: $\boldsymbol{h}_Q^{(l+1)} \leftarrow \text{Propg}(\boldsymbol{h}_{FF}^{(l)}, \mathcal{E})$ in Eq. (8)
14: $\boldsymbol{h}_G^{(l+1)} \leftarrow \text{Propg}(\boldsymbol{h}_G^{(l)}, \mathcal{E})$ in Eq. (5)
15: $\boldsymbol{h}_A^{(l)} \leftarrow \text{Propg}(\boldsymbol{h}_{FF}^{(l)}, \mathcal{E}_B)$ in Eq. (10)
16: $\boldsymbol{h}_N^{(l+1)} \leftarrow \text{Propg}(\boldsymbol{h}_A^{(l)}, \mathcal{E}_B)$ in Eq. (9)
17: $\boldsymbol{h}_{FF}^{(l+1)} \leftarrow \text{AGG}(\boldsymbol{h}_G^{(l+1)}, \boldsymbol{h}_Q^{(l+1)}, \boldsymbol{h}_N^{(l+1)})$ in Eq. (11)
18: $l \leftarrow l + 1$
19: $\mathcal{H} \leftarrow \mathcal{H} \cup \boldsymbol{h}_{FF}^{(k)}$
20: **return** $\mathcal{H}$;

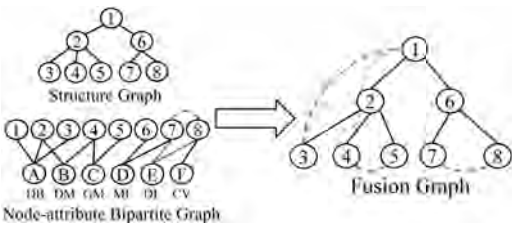


Figure 5: An example of fusion graph.

As described in Section 4, in the training stage, the output set $\mathcal{H}$ is used in the loss function to optimize the model learning. In the online query stage, the output is translated to the predicted communities through the Community Identification process as described below.

6.6 Community Identification

For the attributed community search problem, we need to find vertices having both dense structure and similar attributes to the query. Thus on top of the online query stage described in Section 4.3 which ensures connectivity with the query vertices, we also enhance the connectivity between graph vertices sharing identical attributes by a fusion graph $G_F = \{\mathcal{V}, \mathcal{E}_F\}$ which combines the information of structure graph $G = \{\mathcal{V}, \mathcal{E}\}$ and bipartite graph $G_B = \{\mathcal{V}, \hat{\mathcal{F}}, \mathcal{E}_B\}$.

To build the fusion graph, we link vertices with the same attributes in the structure graph. The connectivity in the fusion graph represents both the structure connectivity and attribute similarity. Then the fusion graph G_F is fed to Algorithm 1 for a constrained BFS with the model output $\boldsymbol{h}_q$ for community identification.

Example. *Figure 5 shows the fusion graph for our running example. We add a dashed blue edge between two vertices in the structure*

graph if they have the same attribute, e.g., vertices 7 and 8 are connected by a dashed blue edge because they both have attribute "DL".

6.7 Complexity Analysis

In order to analyze the time complexity of QD-GNN and AQD-GNN, we first present the complexity of general GNN in Eq.(1). This GNN aggregates neighbors' features for every vertex with the cost of $\sum_{i=\mathrm{u}}^{n} d_{\mathrm{u}}$, where d_{u} is the degree of vertex u and n is number of vertices. Thus the complexity of general GNN is $O(|\mathcal{E}|)$.

For QD-GNN, Query Encoder and Graph Encoder have the same time complexity of $O(|\mathcal{E}|)$ as general GNN. For AQD-GNN, the time cost of Attribute Encoder is also dependent on the sum of vertices' degree in the bipartite graph with the complexity of $O(|\mathcal{E}_B|)$. The aggregation operation in Feature Fusion, e.g., MAX, Concatenation, etc., is implemented in parallel and the complexity is just $O(1)$. Suppose QD-GNN or AQD-GNN is a k-layer model with t iterations, where $k = 3$ and $t = 300$ are typical settings. The complexity of QD-GNN is $O(k \times t \times |\mathcal{E}|)$ in the model training stage and $O(k \times |\mathcal{E}|)$ in the online query stage. Similarly, the complexity of AQD-GNN is $O(k \times t \times (|\mathcal{E}| + |\mathcal{E}_B|))$ in the model training stage and $O(k \times (|\mathcal{E}| + |\mathcal{E}_B|))$ in the online query stage.

7 EXPERIMENTS

In this section, we present our experimental studies to validate the performance of our framework with the three proposed models in different scenarios. We first introduce the setup of our experiment in Section 7.1. Then we evaluate the performance in both attributed and non-attributed community search problem in Section 7.2. To further verify the effectiveness of our models, we compare our models with the interactive community search model ICS-GNN in Section 7.3. Moreover, we evaluate the performance of our model for ACS on large graphs in Section 7.4. Finally, we conduct the ablation study in Section 7.5 to demonstrate the effectiveness of Feature Fusion, the sensitivity test of the parameter γ, and the data split ratio.

7.1 Experimental Setup

7.1.1 Data Sets. To thoroughly evaluate the performance of our framework, we conduct experimental studies on 15 attributed graphs. Table 1 reports the dataset statistics. The first six networks, Cornell, Texas, Washington (Washt), Wisconsin (Wiscs), Cora and Citeseer, are publication citation networks. Each attribute describes the absence/presence of one word in a publication. All these graphs can be found at LINQS website[1]. Reddit [16] is an online discussion website where each vertex is a post and an edge links two posts if they have comments from the same user. Facebook [33] is a social network where vertices are users and edges are friend relationships. It contains 8 ego-networks with different attributes as shown in Table 1. We consider each ego-network as an independent data set. All data sets contain ground-truth communities.

7.1.2 Baseline Models. We compare our models with five state-of-the-art approaches, including two non-attributed community search algorithms: CTC [24] and k-ECC [6], two attributed community search algorithms: ACQ [10] and ATC [22], and a GNN-based interactive community search model ICS-GNN [14].

[1]https://linqs.soe.ucsc.edu/data

Table 1: Dataset Statistics. $|\mathcal{V}|$ and $|\mathcal{E}|$ are the number of vertices and edges. $|\hat{\mathcal{F}}|$ is the number of distinct attributes, K is the number of communities and AS is the average size of communities. Here, M= 10^6.

Data set		$\lvert\mathcal{V}\rvert$	$\lvert\mathcal{E}\rvert$	$\lvert\hat{\mathcal{F}}\rvert$	$\lvert\mathcal{E}_B\rvert$	K	AS
Citation Networks	Cornell	195	283	1703	18496	5	39
	Texas	187	280	1703	15437	5	37.4
	Washt	230	366	1703	19953	5	46
	Wiscs	265	459	1703	25479	5	53
	Cora	2708	5278	1433	49216	7	386.86
	Citeseer	3312	4536	3703	105165	6	552
Social Networks	Reddit	232965	114**M**	602	140**M**	50	4659.3
	FB-0	348	2852	224	3348	24	13.54
	FB-107	1046	27783	576	11827	9	55.67
	FB-1684	793	14810	319	6131	17	45.71
	FB-1912	756	30772	480	8066	46	23.15
	FB-3437	548	5347	262	4263	32	6
	FB-348	228	3416	161	2398	14	40.5
	FB-414	160	1843	105	1566	7	25.43
	FB-686	171	1824	63	999	14	34.64

7.1.3 Query Setting. For each data set, we generate $n_q = 350$ pairs of input query set $\mathcal{Q} = \{< \mathcal{V}_q, \mathcal{F}_q >\}_{q=1}^{n_q}$ and the corresponding ground-truth community $\mathcal{Y}_q$. To generate the query vertex set $\mathcal{V}_q$, vertex sets containing 1-3 vertices are randomly selected from the ground-truth community. To generate the query attribute set $\mathcal{F}_q$, we design three different types as described below for fair comparison with different existing CS and ACS methods. The query vertex set and corresponding ground-truth communities are shared across the three types of input queries.

- **Empty attribute query (EmA)**. To compare with methods for non-attributed community search, we set the attribute query set empty ($\mathcal{F}_q = \emptyset$) and generate the EmA set $\mathcal{Q}_{\mathrm{EmA}} = \{< \mathcal{V}_q, \emptyset >\}$.
- **Attribute from community (AFC)**. As suggested by [10, 22], to construct the query attribute set ($\mathcal{F}_q = \mathcal{F}_q^{\mathrm{c}}$), we use 5 most common attributes in ground-truth communities. Therefore, we have $\mathcal{Q}_{\mathrm{AFC}} = \{< \mathcal{V}_q, \mathcal{F}_q^{\mathrm{c}} >\}$. AFC is used to validate the contribution of the attributes in the community search.
- **Attribute from node (AFN)**. We simulate real queries provided by users and select 5 most common attributes from attributes of query vertices as the query attribute set, i.e, $\mathcal{F}_q = \mathcal{F}_q^{\mathrm{n}}$. In other words, $\mathcal{F}_q^{\mathrm{n}}$ may be unrelated to the ground-truth communities. We construct the AFN set $\mathcal{Q}_{\mathrm{AFN}} = \{< \mathcal{V}_q, \mathcal{F}_q^{\mathrm{n}} >\}$. Obviously, AFN is a more challenging setting and closer to the real scenarios.

7.1.4 Data Splitting. For each data set, we split 350 query-community pairs into training data, validation data and test data with the ratio of 150:100:100 by default. We use training data to train our models, validation data to select the best weights during the training process, and test data to measure the performance of all methods. In the ablation study, we vary the data splitting ratio to evaluate its influence on the performance.

7.1.5 Evaluation Metrics. Let $D_{\mathrm{test}} = \{\mathcal{Q}, \hat{\mathcal{C}}, \mathcal{Y}\}$ be the test data set, where $\mathcal{Q}$ is the query set, $\hat{\mathcal{C}}$ is the predicted community set by a method and $\mathcal{Y}$ is the ground-truth community set. To measure the quality of communities found by different methods, we employ F1-score to evaluate the quality of the predicted set $\hat{\mathcal{C}}$. F1-score is defined as:

$$F1(\hat{\mathcal{C}}, \mathcal{Y}) = \frac{2 \cdot pre(\hat{\mathcal{C}}, \mathcal{Y}) \cdot rec(\hat{\mathcal{C}}, \mathcal{Y})}{pre(\hat{\mathcal{C}}, \mathcal{Y}) + rec(\hat{\mathcal{C}}, \mathcal{Y})}$$

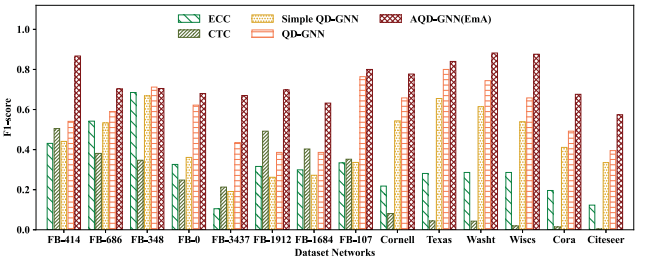


Figure 6: Non-attributed community search performance comparison.

where $pre(\hat{C}, \mathcal{Y})$ is the precision of predicted community set $\hat{C}$ on the ground-truth community set $\mathcal{Y}$, $rec(\hat{C}, \mathcal{Y})$ is the recall of the predicted communities:

$$pre(\hat{C}, \mathcal{Y}) = \frac{\sum_{c_q \in \hat{C}} \boldsymbol{c}_q \& \boldsymbol{y}_q}{\sum_{c_q \in \hat{C}} \sum_{i=0}^{n} \boldsymbol{c}_{qi}}, rec(\hat{C}, \mathcal{Y}) = \frac{\sum_{c_q \in \hat{C}} \boldsymbol{c}_q \& \boldsymbol{y}_q}{\sum_{\boldsymbol{y}_q \in \mathcal{Y}} \sum_{i=0}^{n} \boldsymbol{y}_{qi}}.$$

Here, $\boldsymbol{c}_q \in \{0,1\}^{n \times 1}$ and $y_q \in \{0,1\}^{n \times 1}$ are the predicted and ground-truth community vectors for query q respectively.

7.1.6 Implementation Details. In our models, we build three layers with 128 neurons in the hidden layer. We train 300 iterations with a learning rate of 0.001. In the Feature Fusion component, we choose concatenate as the aggregation function in Eq. (6) and Eq. (11). In each layer except the output layer, we employ ReLU as activation function, batch normalization with batch size 4 and dropout rate 0.5 [38] for each branch.

7.2 Community Search Performance

We present comprehensive experiments to validate the query performance of the three proposed models under two settings: non-attributed community search, and attributed community search.

7.2.1 Non-attributed community search. In order to compare to non-attributed community search algorithms, we generate the multi-vertex queries set without query attributes Q_{EmA}, and compare our three models Simple QD-GNN, QD-GNN, AQD-GNN with CTC and k-ECC. Figure 6 shows the F1-score. We can observe that:

- CTC performs reasonably well in Facebook ego networks but poorly in citation networks, since CTC searches communities using the k-truss subgraph pattern, which may fit the dense social networks well, but does not fit the sparser citation networks.
- By capturing the local query structure only, Simple QD-GNN can outperform ECC and CTC in citation networks and achieve comparable performance in Facebook ego networks. It demonstrates the learning-based models can apply to different types of networks and discover communities with different structural properties.
- QD-GNN can substantially outperform Simple QD-GNN by improving the F1-score by 0.14 on average. It validates the effectiveness of the query-independent graph features learned from Graph Encoder.
- We also apply AQD-GNN to non-attributed community search, where we set the query attribute set to empty, $\mathcal{F}_q = \emptyset$. Interestingly, AQD-GNN can achieve the best performance in almost all data sets in Figure 6. This is owing to the Feature Fusion operator and Attribute Encoder design in AQD-GNN. Specifically, Feature Fusion can transmit graph information and query vertices information to Attribute Encoder before the second layer. Then Attribute Encoder can utilize the information from the second layer and learn hidden relations between attributes.

7.2.2 Attributed community search. We compare AQD-GNN with two attributed community search algorithms: ACQ and ATC. ACQ can only handle one query vertex while ATC and our model AQD-GNN can handle multiple query vertices. Thus we compare ACQ and AQD-GNN for one-vertex queries in Figure 7a, and compare ATC and AQD-GNN for multi-vertex queries in Figure 7b. We can observe that:

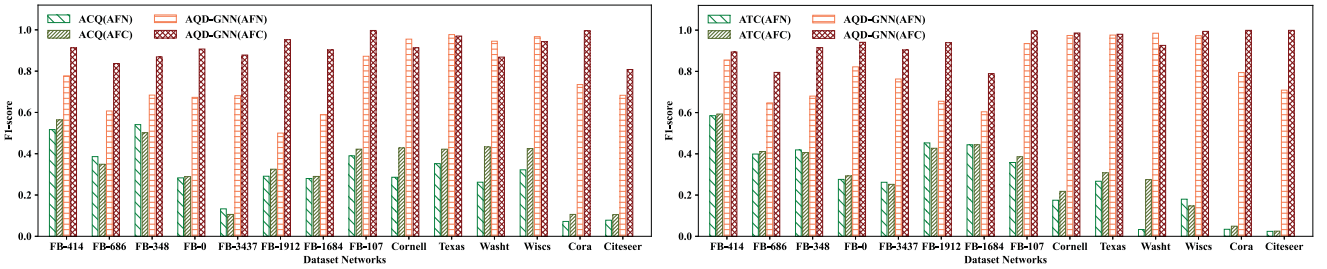
- For Cora and Citeseer with large ground-truth communities (hundreds of vertices in a community), the performances of ATC and ACQ are quite poor (around 0.1 in F1-score). It is because their pre-defined community patterns (i.e., k-core and k-truss) are too strict to find large communities in the real-world graphs.
- AQD-GNN consistently performs the best on all data sets. As a data-driven approach, AQD-GNN is capable of learning communities with varied sizes and shapes. It performs stably on all graphs benefiting from learning adaptive weight matrices for different data sets.
- Compared to AFC, all methods suffer from performance degradation under the AFN setting in most data sets, since AFC is a more favorable setting where the query attribute set is directly extracted from the most common attributes of the ground-truth. For example, in the Washington data set in Figure 7b, ATC achieves 0.275 F1-score under AFC, but only 0.033 under AFN.
- Under the more realistic but challenging AFN setting, we can observe AQD-GNN achieves a significant performance improvement over the baselines, with 0.46 and 0.53 improvements on F1-score for one-vertex queries and multi-vertex queries respectively. This is because AQD-GNN exploits the node-attribute bipartite graph to find similar attributes, while the baselines simply require vertices in a community have identical attributes with query attributes.

7.2.3 Query Efficiency. We evaluate the query efficiency of AQD-GNN in the test set. Table 2 shows the average query time (in milliseconds) of 100 test queries by AQD-GNN and baselines. The last column reports the average query time among all data sets.

Overall, the query time AQD-GNN is much faster than that of all baselines except ACQ. ACQ is a simple baseline which only allows one query vertex and considers vertices' degrees and common attributes. Its query performance in terms of F1-score is very poor as shown in Figure 7. It is worth noting that AQD-GNN achieves a stable query time of around 5 milliseconds on all data sets, while the query time of CTC, ECC and ATC increases significantly when the graph is large. In particular, CTC takes almost 5,000 milliseconds for a query on FB-1912, while AQD-GNN only costs 4.96 milliseconds. This experiment shows that AQD-GNN is more suitable for online search in real-world applications.

7.3 Interactive Community Search

ICS-GNN [14] is a recent GNN-based model for interactive community search. Given a query, ICS-GNN returns an answer community. If the user is not satisfied with the answer, he/she can give a feedback (e.g., adding some additional vertices), and then ICS-GNN will respond with a revised answer. This interaction continues until



(a) Compared with methods supporting one-vertex queries. **(b) Compared with methods supporting multi-vertex queries.**

Figure 7: Attributed community search performance compared with other approaches.

Table 2: Average query time (in milliseconds) of different community search methods.

Methods		FB-414	FB-686	FB-348	FB-0	FB-3437	FB-1912	FB-1684	FB-107	Cornell	Texas	Washt	Wiscs	Cora	Citeseer	Average
Non-Attributed	CTC	34.78	41.41	120.92	49.25	131.67	4903.38	604.67	2498.82	0.45	0.40	0.42	0.63	1.96	1.28	599.00
	ECC	3.52	2.29	5.20	2.57	6.60	154.23	26.85	93.62	0.24	0.28	0.23	0.33	2.67	1.76	21.50
Attributed	ACQ	<0.01	<0.01	1.45	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01	0.10
	ATC	4.40	5.10	7.70	5.90	10.30	43.60	22.70	40.10	7.90	14.02	2.11	18.60	11.49	3.68	9.80
	AQD-GNN	3.31	3.32	3.41	3.63	4.32	4.96	4.56	5.46	4.15	4.10	4.16	4.41	5.54	5.32	4.31

Table 3: F1-score (in %) and time cost (in seconds) of interactive community search methods on different networks.

Method	FB-414		FB-686		FB-348		FB-0		FB-3437		FB-1912		FB-1684		FB-107		Cora		Citeseer		Reddit		Average	
	F1	Time	F1	Time	F1	Time	F1	Time	F1	Time	F1	Time	F1	Time	F1	Time	F1	Time	F1	Time	F1	Time	F1	Time
ICS-GNN	56.63	0.14	43.53	0.15	33.88	0.26	24.94	0.26	26.49	0.51	20.23	2.36	20.76	1.28	36.46	2.40	30.52	0.14	30.29	0.14	19.41	1.84	31.19	0.86
QD-GNN	**62.02**	0.14	**44.28**	0.14	34.74	0.26	31.07	0.27	27.38	0.52	21.10	2.45	24.79	1.31	38.39	2.49	32.56	0.12	31.53	0.12	21.29	1.68	$33.56_{+2.37}$	$0.87_{+0.01}$
AQD (AFN)	61.25	0.14	43.05	0.14	**35.80**	0.25	31.27	0.26	29.03	0.50	20.44	2.29	36.51	1.23	41.07	2.41	31.81	0.14	**33.09**	0.12	20.71	1.85	$34.91_{+3.72}$	$0.85_{-0.01}$
AQD (AFC)	57.34	0.14	38.87	0.14	35.35	0.25	**35.63**	0.26	**30.22**	0.50	**37.52**	2.29	**37.91**	1.23	**49.67**	2.41	**33.19**	0.14	31.77	0.12	**24.86**	1.85	$\mathbf{37.48}_{+6.29}$	$0.85_{-0.01}$

the user is satisfied. In each interaction, ICS-GNN first finds a candidate subgraph, learns the vertex embedding through a Vanilla GCN model [27] and finally employs a BFS based algorithm to select k-sized community with the maximum GNN scores. Note that ICS-GNN does not use any training queries with ground-truth communities to train the model; for each user query, it re-trains the GNN model to obtain the vertices' embeddings only from the knowledge of the given query. ICS-GNN only supports non-attributed community search.

In this experiment, we replace the Vanilla GCN model [27] in the ICS-GNN [14] with our community search models QD-GNN and AQD-GNN to compare the performance of interactive community search problem.

7.3.1 Performance in Effectiveness. We first use QD-GNN to replace the GNN model in ICS-GNN framework for non-attributed community search. As shown in Table 3, QD-GNN outperforms the original ICS-GNN in all data sets with 2.37% improvement in F1-score. We also use AQD-GNN to replace the GNN model in ICS-GNN so it supports interactive attributed community search. As shown in Table 3, AQD-GNN further improves the F1-score of the original ICS-GNN for all data sets by 3.72% (AFN) and 6.29% (AFC) on average. This experiment proves that our QD-GNN and AQD-GNN models are more effective than Vanilla GCN in the ICS-GNN framework.

7.3.2 Performance in Efficiency. We report the average time of community search per interaction by ICS-GNN, QD-GNN and AQD-GNN in Table 3. The running time of the three models are very close. Without increasing the time cost, we improve the performance of ICS-GNN and extend it to support attributed interactive community search problem.

Table 4: The performances of ACS methods on large data sets.

Methods	Reddit			Enlarged_Reddit		
	Index/Train Time	Query Time	F1-score	Index/Train Time	Query Time	F1-score
ACQ	**42.4 s**	32.2 ms	0.53	**852.7 s**	5726.6 ms*	0.38
ATC #	-	-	-	-	-	-
AQD-GNN	4993.6 s	**6.7 ms**	**0.91**	3898.5 s	**5.3 ms**	**0.91**

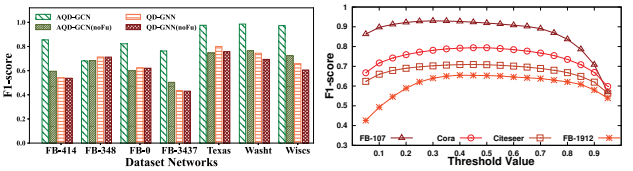
* 25 out of 100 queries are out of memory when processing. The query time are the average of the rest 75 queries.
\# ATC did not finish building its index in 7 days on both two data sets.

7.4 ACS on Large Graphs

In this experiment, we evaluate the performance of our model for ACS on large graphs. We design a subgraph training mechanism to train our models on large graphs. We first select neighbors of query vertices as the candidate subgraph for each query. According to the number of neighbors, we select 1 or 2-hop neighbors in the fusion graph described in Section 6.6. Then we train our model on these small subgraphs and predict communities.

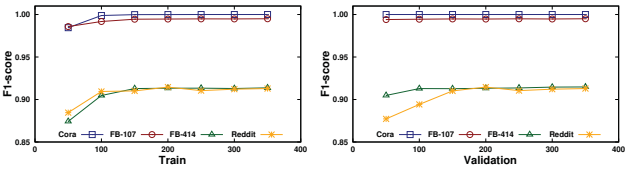
We compare AQD-GNN with ACQ and ATC for attributed community search on Reddit and an enlarged version of Reddit, denoted as Enlarged_Reddit. To enlarge Reddit and preserve the ground-truth communities at the same time, we add some new vertices for edges within a community. A new vertex is linked to the two ends of an edge, and the attributes of the new vertex are the average attribute values of the two ends. The Enlarged_Reddit has 3.12M vertices and 126M edges.

Table 4 reports the index/training time, query time and F1-score of the discovered communities. ACQ takes only 42.4 seconds and 852.7 seconds to build index on Reddit and Enlarge_Reddit. But in terms of the query time, it costs 32.2 milliseconds and 5726.6 milliseconds, while AQD-GNN only costs 6.7 milliseconds and 5.3



(a) F1-score w/wo Feature Fusion. (b) F1-score by varying γ.

Figure 8: Ablation studies for Feature Fusion and γ.



(a) Vary training set size. (b) Vary validation set size.

Figure 9: Ablation study for data split ratio.

milliseconds respectively. It is worth noting that ACQ runs out of memory for 25 out of 100 queries on a 300GB memory server. This is because ACQ finds a k-core community with the largest k containing the query vertices. The k-core community can be quite large, for example, for a query vertex, ACQ first finds a 2-core community with more than 800 thousand candidate vertices. The average F1-score of ACQ is much lower than that of our method in both data sets. ATC did not finish building its index in 7 days and we treat it as timed out. From this experiment, we can see that AQD-GNN achieves a good balance between training time and query time in large graphs, and its F1-score is the best.

7.5 Ablation Study

In this section, we report the ablation studies of our models, including the effectiveness of Feature Fusion, the sensitivity test of the parameter γ, and the data split ratio in the attributed community search task.

7.5.1 Ablation Study for Feature Fusion. In our model, we use the aggregation result $\boldsymbol{h}_{FF}$ in Eq. (6) in QD-GNN and Eq. (11) in AQD-GNN to fuse all information, and assign fused features to Query Encoder and Attribute Encoder in Eq. (7) and Eq. (12). To verify the effectiveness of Feature Fusion, we compare the original AQD-GNN and QD-GNN models with AQD-GNN-noFu and QD-GNN-noFu where the encoders do not aggregate in the hidden layer. They only aggregate after the last layer to output the final results.

The comparison results are shown in Figure 8a. For the non-attributed community search problem, the effect of Feature Fusion is more significant in citation networks than that in Facebook ego networks. In QD-GNN, Feature Fusion aims to mix the global graph information and local query knowledge. The Facebook ego networks themselves are local graphs. Therefore, Feature Fusion in QD-GNN can only improve the model slightly on Facebook ego networks. For ACS problem, AQD-GNN outperforms AQD-GNN-noFu substantially in both Facebook ego networks and citation networks. This is because Feature Fusion in AQD-GNN not only fuses the global graph feature, local query structure and similar attribute information at the same time, but also processes query vertices and query attributes simultaneously through the updating of Query Encoder and Attribute Encoder by fused features. This fusion and updating operations significantly improve the results.

7.5.2 Ablation Study for the threshold γ. When translating the model output vector h_q from $\mathbb{R}^n$ to the community vertex set $\hat{C}_q$ in Section 4.3, we use a threshold $\gamma \in [0, 1]$ in the constrained BFS in Algorithm 1: if $h_{qi} \geq \gamma$, then vertex $v_i \in \hat{C}$, otherwise $v_i \notin \hat{C}$. In above experiments, we choose γ which achieves the best performance in the validation set. To analyze the impact of the threshold, we vary γ from 0.05 to 0.95 and report the F1-score in Figure 8b. When γ is between 0.3 and 0.7, there is very little fluctuation in the performance. Therefore, AQD-GNN is not very sensitive to the selecting of this threshold γ.

7.5.3 Ablation Study for the data split ratio. In all the experiments above, we fix the training/validation/test size ratio as 150:100:100. To test the sensitiveness of data split ratio, we vary the training set size from 50 to 350, and fix both the validation and test set size as 100. The results are plotted in Figure 9a. We also vary the validation set size and fix the training set size as 150 and the test set size as 100. The results are plotted in Figure 9b.

When the training set size increases from 50 to 100, the F1-score of all data sets has a notable increase, but when the training set size further increases from 100 to 350, the F1-score remains quite stable. When varying the validation set size, for Cora and FB-107 the F1-score remains stable; for FB-414 and Reddit, the F1-score increases when the validation set size increases from 50 to 100, and then remains stable afterwards.

This experiment shows that when the training/validation set is very small, increasing the size can improve the performance; but when the size is above 100, the performance remains stable.

8 CONCLUSIONS

In this paper, we propose the QD-GNN and AQD-GNN for non-attributed community search and attributed community search respectively. In QD-GNN, we first propose a query-driven component to acquire queries directly and avoid the re-training process in the existing GNN-based community search model ICS-GNN. Then we combine the local query-dependent structure and global query-independent vertex embedding. For attributed community search, we model vertex attributes as a bipartite graph and further propose the AQD-GNN model. To the best of our knowledge, AQD-GNN is the first GNN model for attributed community search. Moreover, we apply QD-GNN and AQD-GNN in the framework of ICS-GNN for interactive attributed community search. Experiments demonstrate that the proposed models outperform previous approaches significantly. The proposed models are trained through historical queries (training queries), then applied for online query. In the future, more research on training query selection can be carried out to train the model with limited training queries for large graphs. In addition, as time goes by, more historical queries can be collected and the model can be updated with them as training queries to improve its performance. The model update mechanism is worth further study.

ACKNOWLEDGMENTS

The work was supported by grants from NSFC Grant No. U1936205, the Research Grant Council of the Hong Kong Special Administrative Region, China [Project No.: CUHK 14205618], Tencent AI Lab RhinoBird Focused Research Program GF202101, and CUHK Direct Grant No. 4055159. Additional funding was provided by the HK RGC Grant Nos. 22200320 and 12200021.

REFERENCES

[1] Esra Akbas and Peixiang Zhao. 2017. Truss-based community search: a truss-equivalence based indexing approach. *PVLDB* 10, 11 (2017), 1298–1309.

[2] Aleksandar Bojchevski and Stephan Günnemann. 2019. Adversarial Attacks on Node Embeddings via Graph Poisoning. In *ICML*. 695–704.

[3] Heng Chang, Yu Rong, Tingyang Xu, Yatao Bian, Shiji Zhou, Xin Wang, Junzhou Huang, and Wenwu Zhu. 2021. Not All Low-Pass Filters are Robust in Graph Convolutional Networks. *NeurIPS* 34 (2021).

[4] Heng Chang, Yu Rong, Tingyang Xu, Wenbing Huang, Somayeh Sojoudi, Junzhou Huang, and Wenwu Zhu. 2021. Spectral graph attention network with fast eigen-approximation. In *CIKM*. 2905–2909.

[5] Heng Chang, Yu Rong, Tingyang Xu, Wenbing Huang, Honglei Zhang, Peng Cui, Wenwu Zhu, and Junzhou Huang. 2020. A restricted black-box adversarial framework towards attacking graph embedding models. In *AAAI*, Vol. 34. 3389–3396.

[6] Lijun Chang, Xuemin Lin, Lu Qin, Jeffrey Xu Yu, and Wenjie Zhang. 2015. Index-based optimal algorithms for computing Steiner components with maximum connectivity. In *SIGMOD*. 459–474.

[7] Wanyun Cui, Yanghua Xiao, Haixun Wang, Yiqi Lu, and Wei Wang. 2013. Online search of overlapping communities. In *SIGMOD*. 277–288.

[8] Wanyun Cui, Yanghua Xiao, Haixun Wang, and Wei Wang. 2014. Local search of communities in large graphs. In *SIGMOD*. 991–1002.

[9] Chenhui Deng, Zhiqiang Zhao, Yongyu Wang, Zhiru Zhang, and Zhuo Feng. 2020. GraphZoom: A multi-level spectral approach for accurate and scalable graph embedding. *ICLR* (2020).

[10] Yixiang Fang, Reynold Cheng, Siqiang Luo, and Jiafeng Hu. 2016. Effective community search for large attributed graphs. *PVLDB* 9, 12 (2016), 1233–1244.

[11] Yixiang Fang, Xin Huang, Lu Qin, Ying Zhang, Wenjie Zhang, Reynold Cheng, and Xuemin Lin. 2020. A survey of community search over big graphs. *VLDBJ* 29, 1 (2020), 353–392.

[12] Alex Fout, Jonathon Byrd, Basir Shariat, and Asa Ben-Hur. 2017. Protein interface prediction using graph convolutional networks. In *NeurIPS*. 6530–6539.

[13] Hongyang Gao and Shuiwang Ji. 2019. Graph U-Nets. In *ICML*. 2083–2092.

[14] Jun Gao, Jiazun Chen, Zhao Li, and Ji Zhang. 2021. ICS-GNN: lightweight interactive community search via graph neural network. *PVLDB* 14, 6 (2021), 1006–1018.

[15] Arushi Goel, Keng Teck Ma, and Cheston Tan. 2019. An End-to-End Network for Generating Social Relationship Graphs. In *CVPR*. 11186–11195.

[16] William L Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *NeurIPS*. 1025–1035.

[17] Chaoyang He, Tian Xie, Yu Rong, Wenbing Huang, Junzhou Huang, Xiang Ren, and Cyrus Shahabi. 2019. Cascade-bgnn: Toward efficient self-supervised representation learning on large-scale bipartite graphs. *arXiv preprint arXiv:1906.11994* (2019).

[18] Chaoyang He, Tian Xie, Yu Rong, Wenbing Huang, Yanfang Li, Junzhou Huang, Xiang Ren, and Cyrus Shahabi. [n.d.]. Bipartite graph neural networks for efficient node representation learning. *arXiv preprint arXiv:1906.11994* ([n. d.]).

[19] Jiafeng Hu, Xiaowei Wu, Reynold Cheng, Siqiang Luo, and Yixiang Fang. 2016. Querying minimal steiner maximum-connected subgraphs in large graphs. In *CIKM*. 1241–1250.

[20] Wenbing Huang, Tong Zhang, Yu Rong, and Junzhou Huang. 2018. Adaptive Sampling Towards Fast Graph Representation Learning. In *NeurIPS*. 4558–4567.

[21] Xin Huang, Hong Cheng, Lu Qin, Wentao Tian, and Jeffrey Xu Yu. 2014. Querying k-truss community in large and dynamic graphs. In *SIGMOD*. 1311–1322.

[22] Xin Huang and Laks VS Lakshmanan. 2017. Attribute-driven community search. *PVLDB* 10, 9 (2017), 949–960.

[23] Xin Huang, Laks VS Lakshmanan, and Jianliang Xu. 2017. Community search over big graphs: Models, algorithms, and opportunities. In *ICDE*. 1451–1454.

[24] Xin Huang, Laks VS Lakshmanan, Jeffrey Xu Yu, and Hong Cheng. 2015. Approximate closest community search in networks. *PVLDB* 9, 4 (2015), 276–287.

[25] Xin Huang, Laks V. S. Lakshmanan, and Jianliang Xu. 2019. *Community Search over Big Graphs*. Morgan & Claypool Publishers.

[26] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *ICML*. PMLR, 448–456.

[27] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *ICLR*.

[28] Junhyun Lee, Inyeop Lee, and Jaewoo Kang. 2019. Self-Attention Graph Pooling. In *ICML*. 3734–3743.

[29] Jia Li, Yu Rong, Hong Cheng, Helen Meng, Wen-bing Huang, and Junzhou Huang. 2019. Semi-Supervised Graph Classification: A Hierarchical Graph Perspective. In *WWW*. 972–982.

[30] Ye Li, Chaofeng Sha, Xin Huang, and Yanchun Zhang. 2018. Community Detection in Attributed Graphs: An Embedding Approach. In *AAAI*. 338–345.

[31] Hehuan Ma, Yatao Bian, Yu Rong, Wenbing Huang, Tingyang Xu, Weiyang Xie, Geyan Ye, and Junzhou Huang. 2022. Cross-Dependent Graph Neural Networks for Molecular Property Prediction. *Bioinformatics* (2022).

[32] Yao Ma, Suhang Wang, Charu C. Aggarwal, and Jiliang Tang. 2019. Graph Convolutional Networks with EigenPooling. In *KDD*. 723–731.

[33] Julian J. McAuley and Jure Leskovec. 2012. Learning to Discover Social Circles in Ego Networks. In *NeurIPS*. 548–556.

[34] Yu Rong, Yatao Bian, Tingyang Xu, Weiyang Xie, Ying Wei, Wenbing Huang, and Junzhou Huang. 2020. Self-supervised graph transformer on large-scale molecular data. *NeurIPS* 33 (2020), 12559–12571.

[35] Yu Rong, Wenbing Huang, Tingyang Xu, and Junzhou Huang. 2020. DropEdge: Towards Deep Graph Convolutional Networks on Node Classification. In *ICLR*.

[36] Chao Shang, Yun Tang, Jing Huang, Jinbo Bi, Xiaodong He, and Bowen Zhou. 2019. End-to-End Structure-Aware Convolutional Networks for Knowledge Base Completion. In *AAAI*. 3060–3067.

[37] Mauro Sozio and Aristides Gionis. 2010. The community-search problem and how to plan a successful cocktail party. In *KDD*. 939–948.

[38] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *JMLR* 15, 1 (2014), 1929–1958.

[39] Damian Szklarczyk, Andrea Franceschini, Stefan Wyder, Kristoffer Forslund, Davide Heller, Jaime Huerta-Cepas, Milan Simonovic, Alexander Roth, Alberto Santos, Kalliopi P Tsafou, et al. 2015. STRING v10: protein–protein interaction networks, integrated over the tree of life. *Nucleic acids research* 43, D1 (2015), D447–D452.

[40] Xiang Wang, Xiangnan He, Yixin Cao, Meng Liu, and Tat-Seng Chua. 2019. KGAT: Knowledge Graph Attention Network for Recommendation. In *KDD*. 950–958.

[41] Xiao Wang, Meiqi Zhu, Deyu Bo, Peng Cui, Chuan Shi, and Jian Pei. 2020. AM-GCN: Adaptive Multi-channel Graph Convolutional Networks. In *SIGKDD*. 1243–1253.

[42] Rui Ye, Xin Li, Yujie Fang, Hongyu Zang, and Mingzhong Wang. 2019. A Vectorized Relational Graph Convolutional Network for Multi-Relational Network Alignment. In *IJCAI*. 4135–4141.

[43] Junchi Yu, Tingyang Xu, Yu Rong, Yatao Bian, Junzhou Huang, and Ran He. 2021. Graph Information Bottleneck for Subgraph Recognition. In *ICLR*.

[44] Long Yuan, Lu Qin, Wenjie Zhang, Lijun Chang, and Jianye Yang. 2017. Index-based densest clique percolation community search in networks. *TKDE* 30, 5 (2017), 922–935.

[45] Kangfei Zhao, Jeffrey Xu Yu, Hao Zhang, Qiyan Li, and Yu Rong. 2021. A Learned Sketch for Subgraph Counting. In *SIGMOD*. 2142–2155.

[46] Kangfei Zhao, Zhiwei Zhang, Yu Rong, Jeffrey Xu Yu, and Junzhou Huang. 2021. Finding critical users in social communities via graph convolutions. *TKDE* (2021).

[47] Dingyuan Zhu, Ziwei Zhang, Peng Cui, and Wenwu Zhu. 2019. Robust Graph Convolutional Networks Against Adversarial Attacks. In *KDD*. 1399–1407.

HYPER-TUNE: Towards Efficient Hyper-parameter Tuning at Scale

Yang Li[†‡], Yu Shen[†‡], Huaijun Jiang[†‡], Wentao Zhang[†], Jixiang Li[‡], Ji Liu[‡], Ce Zhang[§], Bin Cui[†∇]
[†]Key Laboratory of High Confidence Software Technologies (MOE), School of CS, Peking University
[§]Department of Computer Science, ETH Zürich [‡]AI Platform, Kuaishou Technology
[∇]Institute of Computational Social Science, Peking University (Qingdao)
[†]{liyang.cs, shenyu, jianghuaijun, wentao.zhang, bin.cui}@pku.edu.cn
[‡]lijixiang@kuaishou.com [‡]jiliu@kwai.com [§]ce.zhang@inf.ethz.ch

ABSTRACT

The ever-growing demand and complexity of machine learning are putting pressure on hyper-parameter tuning systems: *while the evaluation cost of models continues to increase, the scalability of state-of-the-arts starts to become a crucial bottleneck*. In this paper, inspired by our experience when deploying hyper-parameter tuning in a real-world application in production and the limitations of existing systems, we propose HYPER-TUNE, an efficient and robust distributed hyper-parameter tuning framework. Compared with existing systems, HYPER-TUNE highlights multiple system optimizations, including (1) automatic resource allocation, (2) asynchronous scheduling, and (3) multi-fidelity optimizer. We conduct extensive evaluations on benchmark datasets and a large-scale real-world dataset in production. Empirically, with the aid of these optimizations, HYPER-TUNE outperforms competitive hyper-parameter tuning systems on a wide range of scenarios, including XGBoost, CNN, RNN, and some architectural hyper-parameters for neural networks. Compared with the state-of-the-art BOHB and A-BOHB, HYPER-TUNE achieves up to 11.2× and 5.1× speedups, respectively.

PVLDB Reference Format:
Yang Li, Yu Shen, Huaijun Jiang, Wentao Zhang, Jixiang Li, Ji Liu, Ce Zhang, and Bin Cui. HYPER-TUNE: Towards Efficient Hyper-parameter Tuning at Scale. PVLDB, 15(6): 1256 - 1265, 2022.
doi:10.14778/3514061.3514071

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/PKU-DAIR/HyperTune.

1 INTRODUCTION

Recently, researchers in the database community have been working on integrating machine learning functionality into data management systems, e.g., SystemML [18], SystemDS [8], Snorkel [54], ZeroER [71], TFX [5, 9], "Query 2.0" [72], Krypton [50], Cerebro [51], ModelDB [66], MLFlow [75], HoloClean [56], NorthStar [37] and EaseML [49]. AutoML systems [25, 52, 74], an emerging type of data system, significantly raise the level of abstractions of building ML applications [11, 45]. While hyper-parameters drive both the efficiency and quality of machine learning applications, automatic hyper-parameter tuning attracts intensive interests from both practitioners and researchers [16, 27, 38, 40, 47, 55, 76, 77], and becomes an indispensable component in many data systems [44, 59, 67].

Proceedings of the VLDB Endowment, Vol. 15, No. 6 ISSN 2150-8097.
doi:10.14778/3514061.3514071

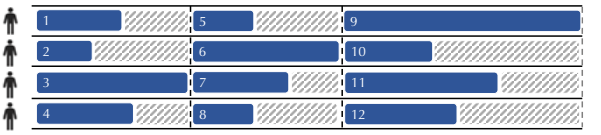


Figure 1: Synchronous mechanism in hyper-parameter tuning, where each row represents a worker. The deep-blue areas correspond to the evaluation process of configurations; the striped areas refer to idle time.

An efficient tuning system, which usually involves sampling and evaluating configurations iteratively, needs to support a diverse range of hyper-parameters, from learning rate, regularization, to those closely related to neural network architectures such as operation types, # hidden units, etc. Automatic tuning methods (e.g., Hyperband [39] and BOHB [16]) have been studied to tune a wide range of models, including XGBoost [12], recurrent neural networks [22], convolutional neural networks [21], etc. In this paper, we focus on building efficient and scalable tuning systems.

Current Landscape. Existing automatic hyper-parameter tuning methods include: Bayesian optimization [7, 24, 62], rule-based search, genetic algorithm [27, 52], random search [6, 15], etc. Many of them have two flavors – *complete evaluation based search* and *partial evaluation based search*. To obtain the performance for each configuration, the complete evaluation based approaches [6, 7, 24] require complete evaluations that are usually computationally expensive. Instead, partial evaluation based methods [16, 34, 36, 39, 42] assign each configuration with incomplete training resources to obtain the evaluation result, thus saving the evaluation resources.

An Emerging Challenge in Scalability. This paper is inspired by our efforts applying these latest methods to applications running at a large Internet company. One critical challenge arises from the **gap between the scalability of existing automatic tuning methods and the ever-growing complexity of industrial-scale models.** In recent years, we have witnessed that evaluating ML models are getting increasingly expensive as the size of datasets and models grows larger. For example, it takes days to train NASNet [78] to convergence on ImageNet, not to mention models like GPT-3 [10] with hundreds of billions of parameters. Unfortunately, it is difficult for existing tuning methods to scale well to such tasks with ever-increasing evaluation costs, thus leading to a sub-optimal configuration for deployment. When deploying existing approaches in

large-scale applications, we realize some limitations in the following three aspects:

(1) Design of Partial Evaluations. Since the complete evaluation of a configuration is usually expensive (e.g., training deep learning models or training ML models on *large-scale* datasets), recent studies propose to evaluate configurations using partial resources (e.g., training models using a few epochs or a subset of the training set) [28, 39, 53]. However, *how many training resources should be allocated to each partial evaluation?* This question is non-trivial: (1) evaluations with small training resources could decrease evaluation cost, however, may be inaccurate to guide the search process; whereas (2) over-allocating resources could have the risk of high evaluation costs but diminishing returns from precision improvements. *How can we automatically decide on the right level of resource allocation to balance the "precision vs. cost" trade-off in partial evaluations?* This question remains open in state-of-the-arts [16, 42, 64].

(2) Utilization of Parallel Resources. Along with the rapid increase of evaluation cost, it comes with the rise of computation resources made available by industrial-scale clusters. However, state-of-the-arts, such as BOHB [16] and MFES-HB [42], often use a *synchronous* architecture, which often cannot fully utilize all computation resources due to the synchronization barrier and are often sensitive to stragglers (See Figure 1). ASHA [40] is able to remove these issues associated with synchronous promotions by incurring a number of inaccurate promotions, while this asynchronous promotion could hamper the sample efficiency when utilizing the parallel and distributed resources. *Thus, we need to explore an efficient asynchronous mechanism which pursues both sample efficiency and high utilization of parallel resources simultaneously.*

(3) Support of Advanced Multi-fidelity Optimizers. While there are recent advancements in the design of Bayesian optimization methods, most, if not all, distributed tuning systems [16, 19, 40] have not fully utilized these advanced algorithms. For example, while there are algorithms that can more effectively exploit the low-fidelity measurements generated by partial evaluations [42], many existing systems [16, 17] still depend on vanilla Bayesian optimization methods that only use the high-fidelity measurements from the complete evaluations. *Can we design a flexible system architecture to conveniently support drop-in replacement of different optimizers under the async/synchronous parallel settings?* This question is especially important from a system perspective.

Contributions. Inspired by our experience and observations deploying these state-of-the-art methods in our scenarios, in this paper, **(C.1)** we propose Hyper-Tune, an efficient distributed automatic hyper-parameter tuning framework. Hyper-Tune has three core components: *resource allocator*, *evaluation scheduler*, and *generic optimizer*, each of which corresponds to one aforementioned question: (1) To accommodate the first issue, we design a simple yet novel resource allocation method that could search for a good allocation via trial-and-error, and this method can automatically balance the trade-off between the *precision* and *cost* of evaluations. (2) To mitigate the second issue, we propose an efficient asynchronous mechanism – D-ASHA, a novel variant of ASHA [40]. D-ASHA pursues the following two aspects simultaneously: (i) *synchronization efficiency*: the overhead of synchronization in wall-clock time, and (ii) *sample efficiency*: the number of runs that the algorithm needs to converge. (3) To tackle the third issue, we provide a modular design that allows us to plug in different hyper-parameter tuning optimizers. This flexible design allows us to plug in MFES-HB [42], a recently proposed multi-fidelity optimizer. In addition, we also adopt an algorithm-agnostic sampling framework, which enables each optimizer algorithm to adapt to the sync/asynchronous parallel scenarios easily. **(C.2)** We conduct extensive empirical evaluations on both publicly available benchmark datasets and a large-scale real-world dataset in production. Hyper-Tune achieves strong anytime and converged performance and outperforms state-of-the-art methods/systems on a wide range of hyper-parameter tuning scenarios: (1) XGBoost with nine hyper-parameters, (2) ResNet with six hyper-parameters, (3) LSTM with nine hyper-parameters, and (4) neural architectures with six hyper-parameters. Compared with the state-of-the-art BOHB [16] and A-BOHB [64], Hyper-Tune achieves up to 11.2× and 5.1× speedups, respectively. In addition, it improves the AUC by 0.87% in an industrial recommendation application with a billion instances.

2 RELATED WORK

Bayesian optimization (BO) has been successfully applied to hyper-parameter tuning [7, 24, 26, 62, 74]. Instead of using complete evaluations, Hyperband [39] (HB) dynamically allocates resources to a set of random configurations, and uses the successive halving algorithm [28] to stop badly-performing configurations in advance. BOHB [16] improves HB by replacing random sampling with BO. Two methods [14, 36] propose to guide early-stopping via learning curve extrapolation. Vizier [19], Ray Tune [46] and OpenBox [43] also include a median stopping rule to stop the evaluations early. In addition, multi-fidelity methods [4, 13, 23, 34, 42, 64] also exploit the low-fidelity measurements from partial evaluations to guide the search for the optimum of objective function f. MFES-HB [42] combines HB with multi-fidelity surrogate based BO.

Many methods [3, 20, 31] can evaluate several configurations in parallel instead of sequentially. However, most of them [20], including BOHB [16], focus on designing batches of configurations to evaluate at once, and few support asynchronous scheduling. ASHA [40] introduces an asynchronous evaluation paradigm based on successive halving algorithm [28]. In addition, Many approaches [1, 32] with asynchronous parallelization cannot exploit multiple fidelities of the objective; A-BOHB [64] supports asynchronous multi-fidelity hyper-parameter tuning. Searching architecture hyper-parameters for neural networks is a popular tuning application. Recent empirical studies [15, 61] show that sequential Bayesian optimization methods [33, 48, 57, 69] achieve competitive performance among a number of NAS methods [2, 47, 55, 60, 73, 78], which highlights the essence of parallelizing these BO related methods.

A-BOHB [64] is the most related method compared with Hyper-Tune, while it suffers from the first issue. BOHB [16] lacks design to tackle the aforementioned three problems, and MFES-HB [42] also faces these first and second issues. Instead, Hyper-Tune is designed to accommodate the three issues simultaneously.

3 PRELIMINARY

We define the hyper-parameter tuning as a black-box optimization problem, where the objective value $f(\boldsymbol{x})$ (e.g., validation error)

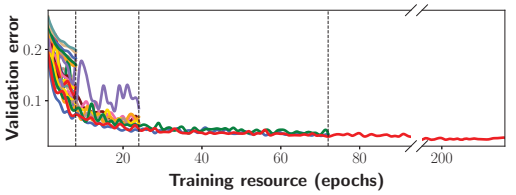


Figure 2: One iteration of successive halving algorithm (SHA) when tuning a CNN on MNIST, where $n_1 = 27$, $r_1 = 1$, $R = 27$, $\eta = 3$, and one unit of resource corresponds to 8 epochs. First, 27 configurations are evaluated with 1 unit of resource, i.e., 8 epochs ($n_1 = 27$ and $r_1 = 1$). Then the top η^{-1} configurations continue their evaluations with η times units of resources (i.e., $n_2 = 27 * \eta^{-1} = 9$ and $r_2 = r_1 * \eta = 3$). Finally, only one configuration is evaluated with the maximum resource R.

reflects the performance of an ML algorithm with given hyper-parameter configuration $\boldsymbol{x} \in \mathcal{X}$. The goal is to find the optimal configuration that minimizes the objective function $\boldsymbol{x}^* = \arg\min_{\boldsymbol{x}\in\mathcal{X}} f(\boldsymbol{x})$, and the only mode of interaction with f is to evaluate the given configuration $\boldsymbol{x}$. In the following, we introduce existing methods for solving this black-box optimization problem, and these methods are the basic ingredients in HYPER-TUNE.

3.1 Bayesian Optimization

The main idea of Bayesian optimization (BO) is as follows. Since evaluating the objective function f for configuration $\boldsymbol{x}$ is very expensive, it approximates f using a probabilistic surrogate model $M : p(f|D)$ that is much cheaper to evaluate. In the n^{th} iteration, BO methods iterate the following three steps: (1) use the surrogate model M to select a configuration that maximizes the acquisition function $\boldsymbol{x}_n = \arg\max_{\boldsymbol{x}\in\mathcal{X}} a(\boldsymbol{x}; M)$, where the acquisition function is used to balance the exploration and exploitation; (2) evaluate the configuration $\boldsymbol{x}_n$ to get its performance y_n; (3) add this measurement $(\boldsymbol{x}_n, y_n)$ to the observed measurements $D = \{(\boldsymbol{x}_1, y_1), ..., (\boldsymbol{x}_{n-1}, y_{n-1})\}$, and refit the surrogate M on the augmented D. Popular acquisition functions include EI [29], PI [62], UCB [63], etc. Due to the ever-increasing evaluation cost, several researches [16, 68] reveal that vanilla BO methods with complete evaluations fail to converge to the optimal configuration quickly.

3.2 Hyperband

To address the issue in vanilla BO methods, Hyperband (HB) [39] proposes to speed up configuration evaluations by early stopping the badly-performing configurations. It has the following two loops:

(1) *Inner loop: successive halving.* HB extends the original successive halving algorithm (SHA) [28], which serves as a subroutine in HB, and here we also refer to it as SHA. SHA is designed to identify and terminate poor-performing hyper-parameter configurations early, instead of evaluating each configuration with complete training resources, thus accelerating configuration evaluation. Given a kind of training resource (e.g., the number of iterations, the size of training subset, etc.), SHA first evaluates n_1 hyper-parameter configurations with the initial r_1 units of resources each, and ranks them by the evaluation performance. Then it promotes the top $1/\eta$ configurations to continue its training with η times larger resources

Table 1: The values of n_i and r_i in the HB evaluations, where $R = 27$ and $\eta = 3$. Each column shows an inner loop (SHA process). The pair (n_i, r_i) in each cell indicates n_i configuration evaluations with r_i units of training resources. Taking the first column "Bracket-1" as an example, the evaluation process corresponds to the iteration of SHA in Figure 2.

	Bracket-1		Bracket-2		Bracket-3		Bracket-4	
i	n_i	r_i	n_i	r_i	n_i	r_i	n_i	r_i
1	27	1	12	3	6	9	4	27
2	9	3	4	9	2	27		
3	3	9	1	27				
4	1	27						

(usually $\eta = 3$), that's, $n_2 = n_1 * \eta^{-1}$ and $r_2 = r_1 * \eta$, and stops the evaluations of the other configurations in advance. This process repeats until the maximum training resource R is reached. Figure 2 gives a concrete example of SHA.

(2) *Outer loop: the choice of r_1 and n_1.* Given some finite budget B for each bracket, the values of r_1 and $n_1 = \frac{B}{r_1}$ should be carefully chosen because a small initial training resource r_1 with a large n_1 may lead to the elimination of good configurations in SHA iterations by mistake. There is no prior whether we should use a smaller initial training resource r_1 with a larger n_1, or a larger r_1 with a smaller n_1. HB addresses this problem by enumerating several feasible values of r_1 in the outer loop, where the inner loop corresponds to the execution of SHA. Table 1 shows a concrete example about the number of evaluations and their corresponding training resources in an iteration of HB, where each column corresponds to the results of inner loop (i.e., one iteration of SHA with different r_1s). For example, the first column "Bracket-1" of Table 1 corresponds to the execution process of SHA in Figure 2. Note that, *the HB iteration will be called multiple times until the tuning budget exhausts.*

Definitions. We refer to SHA with different initial training resources – r_1s as *brackets* (See Table 1), and the evaluations with certain units of training resources as *resource level.*

Partial Evaluation Design Issue in HB. Since HB-style methods [16, 39, 42] own excellent features, such as flexibility, scalability, and ease of parallelization, we build our framework based on HB. HB consists of multiple brackets (i.e., SHA procedures), and each of them requires an r_1 as input. HB enumerates several feasible values of r_1, and executes each corresponding bracket sequentially and repeatedly. Bracket-i is equipped with $r_1 = \eta^{i-1}$ units of initial training resources, so each bracket corresponds to a kind of partial evaluation design. When digging deeper into the HB framework, we observe the "precision vs. cost" tradeoff caused by the selection of bracket (i.e., each kind of partial evaluation design) as follows: (1) The partial evaluation with a small r_1 implies that few training resources are allocated, and this may incur a larger number of inaccurate promotions in SHA, i.e., poor configurations are promoted to the next resource level, and good configurations are terminated by mistake due to the low fidelity. (2) As r_1 becomes large, the partial evaluation design has the risk of high evaluation cost but diminishing returns from precision improvements. While HB tries each bracket sequentially and repeatedly, it is inevitable that it wastes evaluation cost when applying a large number of inappropriate brackets during optimization. To develop an efficient tuning system, we need to revisit the HB pipeline and answer the

following question: *Can we automatically learn the right level of resource allocation (i.e., proper partial evaluation design) that balances the "precision vs. cost" tradeoff well?* In Section 4.1 we describe our bracket selection based solution to this problem.

4 PROPOSED FRAMEWORK

In this section, we first give the overview of the proposed framework, and then describe three core components that are designed to accommodate the aforementioned three issues in Section 1.

Framework Overview. The proposed framework – Hyper-Tune takes the tuning task and time budget as inputs, and outputs the best configuration found in the search process. Hyper-Tune has three components: resource allocator (Section 4.1), evaluation scheduler (Section 4.2), and multi-fidelity optimizer (Section 4.3). It is an iterative framework that will repeat until the given budget exhausts. Figure 3 illustrates an iteration of Hyper-Tune, with four concrete steps. The resource allocator selects the initial training resources r_1 when evaluating configurations (Step 1), which directly determines partial evaluation design. Then the multi-fidelity optimizer will sample a configuration from the search space for each idle worker (Step 2). The evaluation scheduler then evaluates these configurations with the corresponding training resources in parallel (Step 3). Finally, based on the multi-fidelity results from parallel evaluations, Hyper-Tune updates the parameters in resource allocator and multi-fidelity optimizer (Step 4), respectively.

Basic Setting: Measurements and Base Surrogates. Due to the flexibility and scalability of HyperBand (HB) [16, 39, 40, 64], we build our framework on HB. Then we collect the results from evaluations with different resource levels, and we refer to them as "measurements". According to the number of training resources used by the evaluations, we can categorize the measurements into K groups: $D_1, ..., D_K$, where $K = \lfloor \log_\eta(R) \rfloor + 1$, η is the discard proportion in HB, R is the maximum training resources for evaluation, and typically K is less than 7. The measurement $(\boldsymbol{x}, y)$ in each group D_i with $i \in [1 : K]$ is obtained by evaluating configuration $\boldsymbol{x}$ with $r_i = \eta^{i-1}$ units of training resources. Thus D_K denotes the high-fidelity measurements from the complete evaluations with the maximum training resources $r_K = R$, and $D_{1:K-1}$ denote the low-fidelity measurements from the partial evaluations. In Hyper-Tune, we build K base surrogates: $M_{1:K}$, where surrogate M_i is trained on the group of measurements D_i. In the following sections, we introduce the design of each component.

4.1 Resource Allocation with Bracket Selection

The resource allocator aims to design the proper partial evaluations automatically. As stated in Section 3.2, the optimal bracket (i.e., the optimal initial training resources that balance the "precision vs. cost" trade-off well) minimizes the evaluation cost while keeping a high precision of partial evaluations. *We need to automatically deal with this trade-off.* The resource allocator needs to identify the optimal bracket in HB, where each bracket corresponds to a type of initial resource design for partial evaluation.

Solution Overview. We adopt the "trial-and-error" paradigm to identify the optimal bracket in an iterative manner. In each iteration, it iterates the following three steps: (1) we first select a bracket (i.e., partial evaluation design involving n_1 configurations with r_1 initial

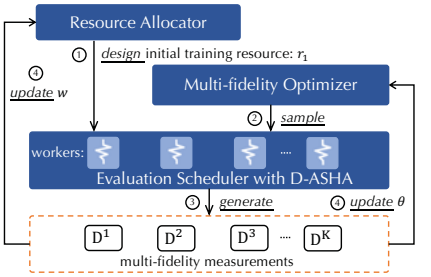


Figure 3: The framework of Hyper-Tune.

training resources) based on parameters $\boldsymbol{w}$; (2) once the i^{th} bracket is chosen, we execute this bracket; (3) based on the measurements from these evaluations, we could update the parameters $\boldsymbol{w}$. *For Step 1, in the beginning, we select brackets by round-robin with three times (as initialization)*; then we sample a bracket using parameters $\boldsymbol{w}$, where each w_i with $i \in [1 : K]$ indicates the probability of this bracket being the optimal one. For Step 3, we propose a two-stage technique to calculate $\boldsymbol{w}$ that balances the above trade-off. In the first stage, we learn a parameter θ_i for each bracket, where θ_i is proportional to the precision of evaluations with the training resources r_i. In the second stage, we multiply each θ_i with a coefficient c_i to obtain the final w_i. This coefficient is inversely proportional to the training resources in the partial evaluation; in this way, the strategy tends to choose the bracket with small training resources. *By the multiplication between c_i and θ_i ($w_i = c_i \cdot \theta_i$), we could balance the "precision vs. cost" trade-off in partial evaluations.*

To measure precision, we focus on the partial orderings of measurements among different resource levels. If configuration $\boldsymbol{x}_1$ performs better than $\boldsymbol{x}_2$ when the training resource is r, given the complete training resource $\boldsymbol{x}_1$ still outperforms $\boldsymbol{x}_2$, indicating that the partial evaluations with r units of training resources are accurate, so we can utilize this to measure the precision of evaluations. To implement this, we utilize the predictions of base surrogate M_i built on D_i, and compare the predictive rankings of configurations with the rankings in D_K. For base surrogates $M_{1:K-1}$, we define the ranking loss as the number of miss-ranked pairs as follows:

$$\mathbb{L}(M_i) = \sum_{j=1}^{N_K} \sum_{k=1}^{N_K} \mathbb{1}((M_i(\boldsymbol{x}_j) < M_i(\boldsymbol{x}_k)) \otimes (y_j < y_k)), \tag{1}$$

where $\otimes$ is the exclusive-or operator, $N_K = |D_K|$, and $(\boldsymbol{x_i}, y_i)$ is the measurement in D_K. For the base surrogate M_K trained on D_K directly, we adopt 5-fold cross-validation to calculate its $\mathbb{L}(M_K)$. Further we define each θ_i as the probability that base surrogate M_i has the least ranking loss. Concretely, we use Markov chain Monte Carlo (MCMC) to learn $\boldsymbol{\theta}$ by drawing S samples: $l_{i,s} \sim \mathbb{L}(M_i)$ for $s = 1, ..., S$ and each surrogate $i = 1, ..., K$, and calculating

$$\theta_i = \frac{1}{S} \sum_{s=1}^{S} \mathbb{1}\left(i = \arg\min_{i'} l_{i',s}\right). \tag{2}$$

To obtain $\boldsymbol{c}$, in Hyper-Tune we simply apply the inverse of the corresponding training resources, i.e., $c_i = 1/r_i$. Finally, we normalize the raw $\boldsymbol{w} = \boldsymbol{c} \circ \boldsymbol{\theta}$ to obtain the final $\boldsymbol{w}$, where $\sum_i w_i = 1$.

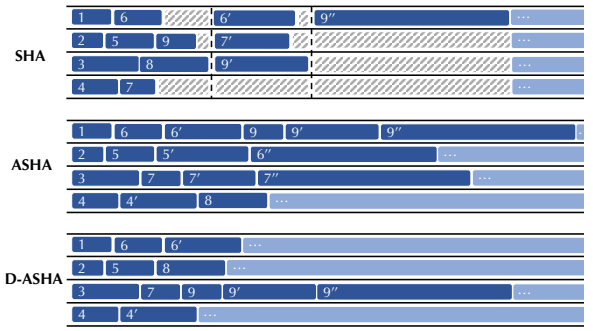


Figure 4: Three scheduling mechanisms on a real-world case, where each row corresponds to a worker, and the ranking of configurations are x_3, x_8, x_2, x_1, x_4, x_5, x_6, x_7, x_9 (latter is the better). i' refers to promoted evaluation of configuration x_i. Each deep-blue block with i corresponds to the evaluation process of x_i; the light-blue blocks represent the evaluations for other iterations of SHA procedures; the striped areas in SHA refer to no evaluations for workers.

4.2 Asynchronous Evaluation Scheduling

In this section, we introduce the distributed scheduling mechanism in the evaluation scheduler. SHA [28] promotes the top $1/\eta$ configurations to the next resource level until all configurations in the current level have been evaluated (synchronization barrier). Due to the synchronous design, which often leads to the straggler issue, the ineffective use of computing resources in SHA is inevitable. ASHA [40] is able to remove these issues associated with synchronous promotions by incurring a number of inaccurate promotions (See Figure 4), i.e., configurations that fall into the top $1/\eta$ early but are not in the actual top $1/\eta$ of all configurations. However, this frequent and inaccurate promotion could hamper the sample efficiency when utilizing the parallel and distributed resources, i.e., ASHA may spend lots of training resources on evaluating the less promising configurations. *Therefore, we need an efficient scheduling method which pursues high sample efficiency while keeping the advantage of asynchronous mechanism.*

Delayed ASHA. To alleviate this issue, we propose a variant of ASHA — delayed ASHA (abbr. D-ASHA), which uses a delay strategy to decrease inaccurate promotions and still preserves the asynchronous scheduling mechanism. Instead of promoting each configuration that is in the top $1/\eta$ of all previously-evaluated configurations, D-ASHA promotes configurations to the next level if (1) the configuration is in the top $1/\eta$ of configurations, and (2) the number of collected measurements $|D_k|$ with current resource level should be η times larger than the number of the next level's $|D_{k+1}|$ if promoted (Lines 9-10 in Algorithm 1). The inaccurate promotions (in Cond.1) arise from a small number of observed measurements in D_k with current resource level. The condition 2 ensures that $|D_k|$ should be larger than a threshold $\eta|D_{k+1}|$, i.e., $|D_k|/(|D_{k+1}|+1) \geq \eta$. In this way, the delay strategy could prevent the frequent promotion issue in ASHA, and further improve the sample efficiency. Figure 4 gives a concrete real-world example to explain this design. Algorithm 1 provides the formulated description about D-ASHA. Additionally, if no promotions are possible, D-ASHA requests a new configuration from the multi-fidelity optimizer (provided in Algorithm 2) and adds it to the base level (Lines 13-14), so that more configurations can be promoted to the upper levels.

Algorithm 1: Pseudo Code for D-ASHA.

Input: initial training resource r_1, maximum resource R, discard proportion η.

1 **Function** D-ASHA():
2 $\boldsymbol{x}, r_x$ = get_job();
3 Assign a job with configuration $\boldsymbol{x}$ and resource r_x to a free worker.
4 **Function** get_job():
5 *// Check if we need to promote configurations.*
6 **for** $k = \lfloor log_\eta(R) \rfloor, ..., 2, 1$, **do**
7 *// D_k refers to measurements of resource level k.*
8 Configuration candidates $C = \{\boldsymbol{x}$ for $\boldsymbol{x} \in$ top $1/\eta$ configurations in D_k if $\boldsymbol{x}$ has not been promoted$\}$
9 **if** $|D_k|/(|D_{k+1}|+1) \geq \eta$ and $|C| > 0$, **then**
10 **return** $C[0], \eta^k$
11 **end if**
12 **end for**
13 Sample a configuration $\boldsymbol{x}$ based on the multi-fidelity optimizer.
14 **return** $\boldsymbol{x}, r_1$

4.3 Multi-fidelity Configuration Sampling

There are various advancements in the design of Bayesian optimization (BO) methods. While those algorithms differ in the execution process, *a flexible tuning system should contain an optimizer module that allows us to plug in different hyper-parameter tuning optimizers easily.* In addition, since most BO based methods are intrinsically sequential, it is impractical to modify each possible algorithm to support parallel scenarios case by case. Thus, we need an algorithm-agnostic framework to extend different sequential optimizers to support parallel evaluations in both sync/asynchronous settings.

Optimizer Design. To tackle the first challenge, we provide a generic optimizer abstraction for configuration sampling in HyperTune. It includes 1) the data structure to store multi-fidelity measurements: D_1, ..., D_K, and 2) the `fit` and `predict` APIs for surrogate model. This abstraction enables convenient support/implementation of different configuration sampling algorithms (e.g., random search, Bayesian optimization, multi-fidelity optimization, etc.). For the second challenge, we further propose an algorithm-agnostic sampling framework to support asynchronous and synchronous parallel evaluations conveniently without any ad-hoc modifications.

(Multi-fidelity Optimizer.) Multi-fidelity methods [23, 30, 35, 41, 53, 58, 70] have achieved success in hyper-parameter tuning. Meanwhile, it produces multi-fidelity measurements which can help determine the optimal bracket for evaluation. In HyperTune, we implement a multi-fidelity optimizer by default based on MFES-HB [41] to utilize multi-fidelity measurements, and build a multi-fidelity ensemble surrogate by combining all base surrogates:

$$M_{\text{MF}} = \texttt{agg}(\{M_1, ..., M_K\}; \boldsymbol{\theta});$$

The surrogate M_{MF} is used to guide the configuration search, instead of the high-fidelity surrogate M_K only, in the framework of BO. Concretely, we combine the base surrogates with weighted bagging, and the weights $\boldsymbol{\theta}$ are exactly the parameters obtained in Section 4.1. Each θ_i also reflects the reliability when applying the corresponding

Algorithm 2: *Sampling* procedure.

Input: the hyper-parameter space $\mathcal{X}$, measurements $D_1, D_2, ..., D_K$, pending configurations C_{pending} being evaluated on workers, the multi-fidelity surrogate M_{MF}, and acquisition function $\alpha(\cdot)$.

1 calculate $\hat{y}$, the median of $\{y_i\}_{i=1}^{n}$ in D_K;
2 impute new measurements $D_{\text{new}} = \{(\boldsymbol{x}_{\text{pending}}, \hat{y}) : \boldsymbol{x}_{\text{pending}} \in C_{\text{pending}}\}$;
3 refit the surrogate M_K in M_{MF} on D_{aug}, where $D_{\text{aug}} = D_K \cup D_{\text{new}}$, and build the acquisition function $\alpha(\boldsymbol{x}, M)$ using M_{MF};
4 **return** the configuration $\boldsymbol{x}^* = \text{argmax}_{\boldsymbol{x}\in\mathcal{X}}\, \alpha(\boldsymbol{x}, M_{MF})$.

low-fidelity information from partial evaluations with r_i units of training resources to the target problem. Finally, the predictive mean and variance of M_{MF} at configuration $\boldsymbol{x}$ are given by:

$$\mu_{MF}(\boldsymbol{x}) = \sum_i \theta_i \mu_i(\boldsymbol{x}), \quad \sigma^2_{MF}(\boldsymbol{x}) = \sum_i \theta_i^2 \sigma_i^2(\boldsymbol{x}), \tag{3}$$

where $\mu_i(\boldsymbol{x})$ and $\sigma_i^2(\boldsymbol{x})$ are the mean and variance predicted by the base surrogate M_i at configuration $\boldsymbol{x}$. Based on the multi-fidelity measurements, this multi-fidelity surrogate could learn the objective function well, and can be used to speed up the search process.

Algorithm-agnostic Sampling. As mentioned previously, we need an algorithm-agnostic framework to extend the sequential method to the sync/asynchronous parallel settings seamlessly. To this end, we adopt an algorithm-agnostic sampling framework, which leverages the local penalization-based strategy [20, 43], where each pending evaluation is imputed with the median of performance in D_K. This framework enables that each algorithm could adapt to the parallel scenarios easily. Algorithm 2 gives the algorithm-agnostic sampling procedure of optimizers.

5 EXPERIMENTS AND RESULTS

We now present empirical evaluations of Hyper-Tune. We first focus on the end-to-end *efficiency* between Hyper-Tune and other state-of-the-art systems. We then study two more specific aspects: scalability and robustness.

5.1 Experimental Settings

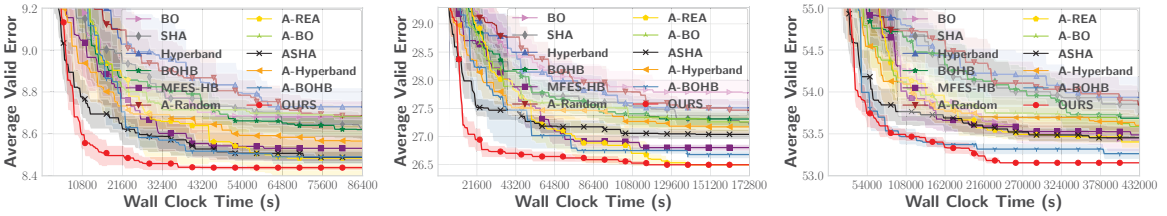
Compared Methods. We compare Hyper-Tune with the manual setting given by our enterprise partner and the following ten baselines. (1) A-Random: Asynchronous Random Search [6] that selects random configurations to evaluate asynchronously, (2) BO: Batch-BO [20] that samples a batch of configurations to evaluate synchronously, (3) A-BO: Async Batch-BO [43] that samples a batch of configurations to evaluate asynchronously, (4) SHA: Successive Halving Algorithm [28] that adaptively allocates training resources to configurations with multi-stage early-stopping, (5) ASHA [40] that improves SHA asynchronously via configuration promotion, (6) Hyperband [39] that applies a bandit strategy to allocate resources dynamically based on SHA, (7) A-Hyperband [40] that extends Hyperband to asynchronous settings via ASHA, (8) BOHB [16] that combines the benefits of both Hyperband and Bayesian optimization, (9) A-BOHB [64] that improves BOHB with asynchronous multi-fidelity optimizations, (10) MFES-HB [42] that combines Hyperband and multi-fidelity Bayesian optimization. Note that Batch-BO, SHA, Hyperband, BOHB, and MFES-HB are synchronous methods, and the others are asynchronous ones.

Tasks. We run experiments on the following tuning tasks:
(1) *Neural Architecture Search.* We use the NAS-Bench-201 [15] which includes offline evaluations of neural network architectures. The search space consists of 6 hyper-parameters. The minimum and maximum number of epochs in NAS-Bench-201 are 1 and 200. HB-based methods use 4 brackets, and the default number of workers is 8. We evaluate Hyper-Tune on three built-in datasets – CIFAR-10-Valid, CIFAR-100, and ImageNet16-120, where the total budgets are 24, 48, and 120 hours, respectively. We finish each method once the simulated training time reaches the given budget.
(2) *Tabular Classification.* We tune XGBoost [12] on four large datasets from OpenML [65] – Pokerhand, Covertype, Hepmass, and Higgs. The hyper-parameter space of XGBoost includes 9 hyperparameters. For partial evaluations, we use the subset of the training set instead of using the entire set. The minimum and maximum size of subset are 1/27 and 1. HB-based methods use 4 brackets, and the default number of workers is 8. The time budgets for the above four datasets are 2, 3, 6, and 6 hours, respectively. Each worker is equipped with 8 CPU cores during evaluation.
(3) *Image Classification.* We tune ResNet [21] on the image classification dataset – CIFAR-10. The search space includes batch size, SGD learning rate, SGD momentum, learning rate decay, and weight decay. Cropping and horizontal flips are used as default augmentation operations. The minimum and maximum number of epochs are 1 and 200. HB-based methods use 4 brackets, and the default number of workers is 4. The time budget is 48 hours. Each worker has 8 CPU cores and 1 GPU during evaluation.
(4) *Language Modelling.* We tune a 3-layer LSTM [22] on the dataset Penn Treebank. The search space includes batch size, hidden size, learning rate, weight decay and five hyper-parameters related to dropout. The embedding size is 400. The minimum and maximum number of epochs are 1 and 200. HB-based methods use 4 brackets, and the time budget is 48 hours; the default number of workers is 4, and each worker uses 8 CPU cores and 1 GPU during evaluation.

Implementation Details. Two metrics are used in our experiments, including (1) the classification error for XGBoost tuning, ResNet tuning, and neural architecture search, and (2) the perplexity when tuning LSTM. We use the validation and test performance stored in NAS-Bench-201 directly for neural architecture search. In the XGBoost tuning experiment, we randomly divide 60% of the total dataset as the training set, 20% as the validation set, and the left as the test set. In the other experiments, we split 20% of the training dataset as the validation set. Then, we track the wall clock time (including optimization overhead and evaluation cost), and store the lowest validation performance after each evaluation. The best configurations are then applied to the test dataset to report the test performance. All methods are repeated ten times with different random seeds, and the mean validation performance across runs is plotted. We include more experimental setups and reproduction details about Hyper-Tune in the supplementary material.

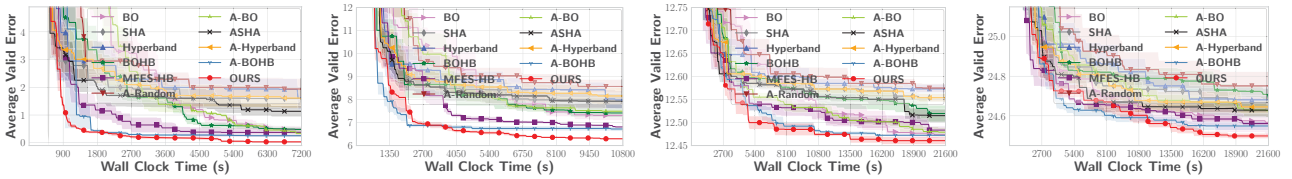
5.2 Architecture Search on NAS-Bench-201

Figure 5 shows the results on NAS-Bench-201 datasets. Due to the utilization of parallel resources issue in Hyperband, asynchronous random search (A-Random) outperforms synchronous Hyperband. Hyper-Tune obtains the best anytime and converged performance among all methods. Concretely, it achieves 8.2×, 11.2× and 6.3× speedups against BOHB, and obtains 3.3×, 2.9×, and 2.0× speedups compared with A-BOHB on CIFAR-10-valid, CIFAR-100,



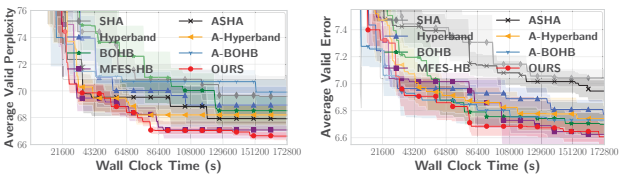
(a) CIFAR-10-valid (b) CIFAR-100 (c) ImageNet16-120

Figure 5: Validation error (%) of tuning architectures on three datasets based on NAS-Bench-201.



(a) Pokerhand (b) Covertype (c) Hepmass (d) Higgs

Figure 6: Validation error (%) of tuning XGBoost on four large datasets.



(a) LSTM on Penn Treebank (b) ResNet on CIFAR-10

Figure 7: Results of tuning LSTM and ResNet.

and ImageNet16-120 respectively, which indicates its *superior efficiency* over the state-of-the-art methods. In addition, Hyper-Tune also gets the best test accuracy (See the results in Appendix A.5).

As reported in NAS-Bench-201 [15], the best method is regularized evolutionary algorithm (REA) [55]. For fair comparison, we also extend REA to an asynchronous version – A-REA. From Figure 5, we have that Hyper-Tune shows consistent superiority over A-REA. Remarkably, Hyper-Tune reaches the global optimum on CIFAR-100 and ImageNet-16-120 across all the ten runs, which also indicates the efficiency of Hyper-Tune.

5.3 Tuning XGBoost on Large Datasets

In Figure 6, we compare Hyper-Tune with the manual setting and ten competitive baselines by tuning XGBoost on four large datasets. The configurations from tuning algorithms outperform the manual settings on test results, which shows the necessity of tuning hyper-parameters for machine learning models. Different from the other experiments, the resource type here is the subset of dataset, i.e., we use different sizes of datasets subset to perform partial evaluation if necessary. As BO and A-BO evaluate each configuration completely, it takes them a long time to converge to a satisfactory performance due to expensive evaluation cost (15 minutes per trial on Covertype). In addition, Hyper-Tune and MFES-HB perform better than HyperBand, BOHB and most asynchronous methods, which indicates the advantage of leveraging low-fidelity measurements. Among the considered methods, Hyper-Tune achieves very competitive anytime performance, and obtains the best converged performance on all of the four datasets.

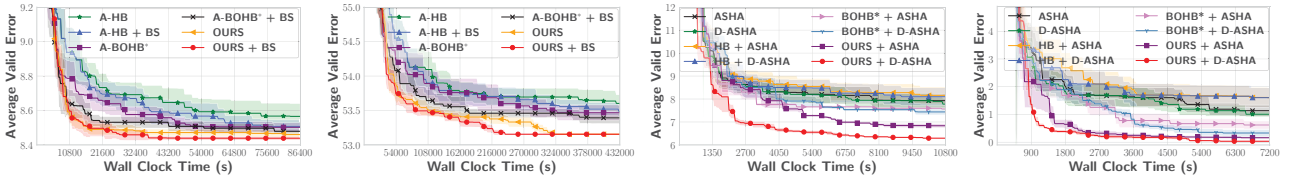
5.4 Tuning LSTM and ResNet

Figure 7(a) show the results of tuning LSTM on Penn Treebank. A-BOHB shows the worst converged performance among baselines, which we attribute to its failure of exploiting multi-fidelity measurements. A-Hyperband, MFES-HB, and Hyper-Tune show similar results in the early stage (19 hours), but after that, the perplexity of A-Hyperband stops decreasing as random sampling fails to exploit history observations efficiently. After 150k secs (about 41 hours), Hyper-Tune outperforms all baselines.

In Figure 7(b), we display the average error of tuning ResNet on CIFAR-10. As SHA and ASHA always start evaluating each configuration from the least resources, they cannot distinguish noisy low-fidelity results, which may explain their overall worst performance. Though Hyper-Tune and MFES-HB obtain a similar result (93.4%), Hyper-Tune shows a better anytime performance due to its asynchronous scheduling.

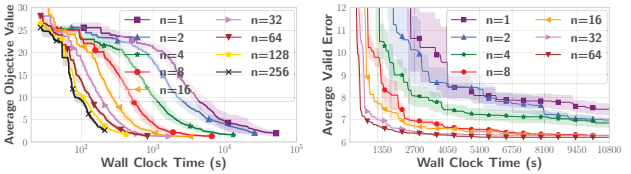
5.5 Scalability Analysis

Figure 9 demonstrates the optimization curve with different number of parallel workers on two tuning tasks. We evaluate Hyper-Tune by tuning the counting-ones function [16] and XGBoost on Covertype. The details about the counting-ones function are provided in Appendix A.4. To demonstrate the scalablility of Hyper-Tune, we set the maximum number of workers to 256 and 64. On both tasks, the anytime performance is better when Hyper-Tune uses more



(a) NAS-Bench-201 on CIFAR-10-valid (b) NAS-Bench-201 on ImageNet16-120 (c) XGBoost on Covertype (d) XGBoost on Pokerhand

Figure 8: Ablation studies for different components in Hyper-Tune.



(a) Counting-ones Benchmark (b) XGBoost on Covertype

Figure 9: Scalability on the number of workers.

workers, which indicates that Hyper-Tune scales to the number of workers well. Notably, Hyper-Tune with the maximum number of workers achieves 145.7x and 18.0x speedups compared with sequential Hyper-Tune on Counting-ones Benchmark and Covertype.

5.6 Industrial-Scale Tuning Application

In addition, we also evaluate Hyper-Tune on an industrial-scale tuning task for recommendation, which aims at identifying active users. The dataset provided by our enterprise partner includes more than one billion instances, and we train the model using the data of seven days and evaluate it using the data of the following day. The number of workers is 10 and the time budget is 48 hours. We evaluate ASHA, BOHB, A-BOHB and Hyper-Tune, and they improve the manual setting by -0.05%, 0.19%, 0.35% and 0.87%, respectively. Moreover, we conduct an ablation study on Hyper-Tune by keeping out one of the component in Table 2. We observe performance gain by introducing each component into Hyper-Tune while Bracket Selection leads to the largest gain. While at least one component is absent in competitive baselines, Hyper-Tune improves the AUC of the second-best baseline A-BOHB by 0.54%, which is a wide margin considering the potential commercial values.

5.7 Ablation Study

Bracket Selection. Figures 8(a) and 8(b) illustrate the effectiveness of the proposed bracket selection method. We also add bracket selection (BS) to the asynchronous variant of Hyperband and BOHB. Note that the asynchronous BOHB here is parallelized via ASHA, but not A-BOHB mentioned in the experimental setups. We have that adding bracket selection helps asynchronous Hyperband, BOHB, and Hyper-Tune converge better. In addition, in Figure 8(b), though the converged performance of Hyper-Tune remains almost the same when bracket selection is employed, the anytime performance improves before 324k secs (90 hours). We owe this gain to the resource allocation strategy learned during optimization rather than attempting all the choices via round robin.

Table 2: Ablation study on Hyper-Tune. The improvement indicates the performance gain upon manual settings.

Methods	Improvement (%)	Δ (%)
w/o BS	0.54	-0.33
w/o D-ASHA	0.75	-0.12
w/o MFES	0.56	-0.31
Hyper-Tune	0.87	-

D-ASHA. Figures 8(c) and 8(d) show the results of applying D-ASHA. For ASHA, Hyperband and BOHB, we observe a slight improvement on both anytime and converged performance when applying D-ASHA. For Hyper-Tune, the validation error decreases by a large margin (0.5%) on Covertype with the aid of D-ASHA. The delay strategy could prevent the frequent promotion issue in ASHA, and further improve the sample efficiency. Therefore, D-ASHA could achieve a higher sample efficiency while keeping the advantage of asynchronous mechanism.

Multi-fidelity Optimizer. We compare different optimizer for configuration sampling, including random sampling (A-Hyperband + BS), high-fidelity optimizer (A-BOHB + BS), and multi-fidelity optimizer (OURS + BS). As shown in Figure 8(a) and 8(b), we have that surrogate-based methods outperform random sampling, while multi-fidelity optimizer outperforms high-fidelity optimizer. The reason is that it takes the low-fidelity measurements into consideration when selecting the next configuration to evaluate. It also indicates that when performing hyper-parameter tuning, the low-fidelity measurements could provide useful information about the objective function, and can be used to speed up the search process.

6 CONCLUSION

In this paper, we presented Hyper-Tune, an efficient and robust distributed hyper-parameter tuning framework at scale. Hyper-Tune introduces three core components targeting at addressing the challenge in the large-scale hyper-parameter tuning tasks, including (1) automatic resource allocation, (2) asynchronous scheduling, and (3) multi-fidelity optimizer. The empirical results demonstrate that Hyper-Tune shows strong robustness and scalability, and outperforms state-of-the-art methods, e.g., BOHB and A-BOHB, on a wide range of tuning tasks.

ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China (No.61832001), Beijing Academy of Artificial Intelligence (BAAI), Kuaishou-PKU joint program, and PKU-Baidu Fund 2019BD006. Bin Cui is the corresponding author.

REFERENCES

[1] Ahsan Alvi, Binxin Ru, Jan-Peter Calliess, Stephen Roberts, and Michael A Osborne. 2019. Asynchronous Batch Bayesian Optimisation with Improved Local Penalisation. In *International Conference on Machine Learning*. PMLR, 253–262.

[2] Noor Awad, Neeratyoy Mallik, and Frank Hutter. 2020. Differential Evolution for Neural Architecture Search. *arXiv preprint arXiv:2012.06400* (2020).

[3] Javad Azimi, Alan Fern, and Xiaoli Z Fern. 2010. Batch bayesian optimization via simulation matching. In *Advances in Neural Information Processing Systems*. 109–117.

[4] Bowen Baker, Otkrist Gupta, Ramesh Raskar, and Nikhil Naik. 2017. Practical neural network performance prediction for early stopping. *arXiv preprint arXiv:1705.10823* 2, 3 (2017), 6.

[5] Denis Baylor, Eric Breck, Heng-Tze Cheng, Noah Fiedel, Chuan Yu Foo, Zakaria Haque, Salem Haykal, Mustafa Ispir, Vihan Jain, Levent Koc, et al. 2017. Tfx: A tensorflow-based production-scale machine learning platform. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 1387–1395.

[6] James Bergstra and Yoshua Bengio. 2012. Random search for hyper-parameter optimization. *Journal of Machine Learning Research* 13, Feb (2012), 281–305.

[7] James S Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. 2011. Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*. 2546–2554.

[8] Matthias Boehm, Iulian Antonov, Sebastian Baunsgaard, Mark Dokter, Robert Erich Ginthoer, Kevin Innerebner, Florijan Klezin, Stefanie Lindstaedt, Arnab Phani, Benjamin Rath, et al. 2020. SystemDS: A Declarative Machine Learning System for the End-to-End Data Science Lifecycle. In *Conference on Innovative Data Systems Research*.

[9] Eric Breck, Neoklis Polyzotis, Sudip Roy, Steven Euijong Whang, and Martin Zinkevich. 2019. Data Validation for Machine Learning. In *3rd Conference on Machine Learning and Systems (MLSys)*.

[10] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165* (2020).

[11] Wenming CAO, Canta ZHENG, Zhiyue YAN, and Weixin XIE. 2022. Geometric deep learning: progress, applications and challenges. *Information Sciences* 65, 126101 (2022), 1–126101.

[12] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 785–794.

[13] Zhongxiang Dai, Haibin Yu, Bryan Kian Hsiang Low, and Patrick Jaillet. 2019. Bayesian Optimization Meets Bayesian Optimal Stopping. (2019), 1496–1506.

[14] Tobias Domhan, Jost Tobias Springenberg, and Frank Hutter. 2015. Speeding Up Automatic Hyperparameter Optimization of Deep Neural Networks by Extrapolation of Learning Curves.. In *IJCAI*, Vol. 15. 3460–8.

[15] Xuanyi Dong and Yi Yang. 2019. NAS-Bench-201: Extending the Scope of Reproducible Neural Architecture Search. In *International Conference on Learning Representations*.

[16] Stefan Falkner, Aaron Klein, and Frank Hutter. 2018. BOHB: Robust and efficient hyperparameter optimization at scale. In *International Conference on Machine Learning*. PMLR, 1437–1446.

[17] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. 2015. Efficient and robust automated machine learning. In *Advances in neural information processing systems*. 2962–2970.

[18] Amol Ghoting, Rajasekar Krishnamurthy, Edwin Pednault, Berthold Reinwald, Vikas Sindhwani, Shirish Tatikonda, Yuanyuan Tian, and Shivakumar Vaithyanathan. 2011. SystemML: Declarative machine learning on MapReduce. In *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 231–242.

[19] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D Sculley. 2017. Google vizier: A service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 1487–1495.

[20] Javier González, Zhenwen Dai, Philipp Hennig, and Neil Lawrence. 2016. Batch bayesian optimization via local penalization. In *Artificial Intelligence and Statistics*. 648–657.

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[22] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[23] Yi-Qi Hu, Yang Yu, Wei-Wei Tu, Qiang Yang, Yuqiang Chen, and Wenyuan Dai. 2019. Multi-fidelity automatic hyper-parameter tuning via transfer series expansion. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 3846–3853.

[24] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. 2011. Sequential model-based optimization for general algorithm configuration. In *International Conference on Learning and Intelligent Optimization*. Springer, 507–523.

[25] Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren (Eds.). 2018. *Automated Machine Learning: Methods, Systems, Challenges*. Springer. In press, available at http://automl.org/book.

[26] Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren. 2019. *Automated machine learning: methods, systems, challenges*. Springer Nature.

[27] Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, et al. 2017. Population based training of neural networks. *arXiv preprint arXiv:1711.09846* (2017).

[28] Kevin Jamieson and Ameet Talwalkar. 2016. Non-stochastic best arm identification and hyperparameter optimization. In *Artificial Intelligence and Statistics*. 240–248.

[29] Donald R Jones, Matthias Schonlau, and William J Welch. 1998. Efficient global optimization of expensive black-box functions. *Journal of Global optimization* 13, 4 (1998), 455–492.

[30] Kirthevasan Kandasamy, Gautam Dasarathy, Jeff Schneider, and Barnabas Poczos. 2017. Multi-fidelity bayesian optimisation with continuous approximations. *arXiv preprint arXiv:1703.06240* (2017).

[31] Kirthevasan Kandasamy, Akshay Krishnamurthy, Jeff Schneider, and Barnabás Póczos. 2017. Asynchronous Parallel Bayesian Optimisation via Thompson Sampling. *stat* 1050 (2017), 25.

[32] Kirthevasan Kandasamy, Akshay Krishnamurthy, Jeff Schneider, and Barnabás Póczos. 2018. Parallelised Bayesian Optimisation via Thompson Sampling. In *International Conference on Artificial Intelligence and Statistics*. 133–142.

[33] Kirthevasan Kandasamy, Willie Neiswanger, Jeff Schneider, Barnabas Poczos, and Eric P Xing. 2018. Neural Architecture Search with Bayesian Optimisation and Optimal Transport. *Advances in Neural Information Processing Systems* 31 (2018).

[34] Aaron Klein, Stefan Falkner, Simon Bartels, Philipp Hennig, and Frank Hutter. 2017. Fast bayesian optimization of machine learning hyperparameters on large datasets. In *Artificial Intelligence and Statistics*. PMLR, 528–536.

[35] Aaron Klein, Stefan Falkner, Simon Bartels, Philipp Hennig, and Frank Hutter. 2017. Fast Bayesian Optimization of Machine Learning Hyperparameters on Large Datasets. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*. 528–536.

[36] Aaron Klein, Stefan Falkner, Jost Tobias Springenberg, and Frank Hutter. 2017. Learning curve prediction with Bayesian neural networks. *Proceedings of the International Conference on Learning Representations* (2017).

[37] Tim Kraska. 2018. Northstar: An Interactive Data Science System. *Proceedings of the VLDB Endowment* 11, 12 (2018).

[38] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. Qtune: A query-aware database tuning system with deep reinforcement learning. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2118–2130.

[39] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. 2018. Hyperband: A novel bandit-based approach to hyperparameter optimization. *Proceedings of the International Conference on Learning Representations* (2018), 1–48.

[40] Liam Li, Kevin Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Jonathan Ben-tzur, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. 2020. A System for Massively Parallel Hyperparameter Tuning. *Proceedings of Machine Learning and Systems* 2 (2020), 230–246.

[41] Yang Li, Jiawei Jiang, Jinyang Gao, Yingxia Shao, Ce Zhang, and Bin Cui. 2020. Efficient Automatic CASH via Rising Bandits. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 4763–4771.

[42] Yang Li, Yu Shen, Jiawei Jiang, Jinyang Gao, Ce Zhang, and Bin Cui. 2021. MFES-HB: Efficient Hyperband with Multi-Fidelity Quality Measurements. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35. 8491–8500.

[43] Yang Li, Yu Shen, Wentao Zhang, Yuanwei Chen, Huaijun Jiang, Mingchao Liu, Jiawei Jiang, Jinyang Gao, Wentao Wu, Zhi Yang, Ce Zhang, and Bin Cui. 2021. OpenBox: A Generalized Black-box Optimization Service. *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining* (2021).

[44] Yang Li, Yu Shen, Wentao Zhang, Jiawei Jiang, Bolin Ding, Yaliang Li, Jingren Zhou, Zhi Yang, Wentao Wu, Ce Zhang, and Bin Cui. 2021. VolcanoML: Speeding up End-to-End AutoML via Scalable Search Space Decomposition. *Proceedings of VLDB Endowment* 14 (2021), 2167–2176.

[45] Zechao Li and Jinhui Tang. 2021. Semi-supervised local feature selection for data classification. *Science China Information Sciences* 64, 9 (2021), 1–12.

[46] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E Gonzalez, and Ion Stoica. 2018. Tune: A Research Platform for Distributed Model Selection and Training. *arXiv preprint arXiv:1807.05118* (2018).

[47] Hanxiao Liu, Karen Simonyan, and Yiming Yang. 2018. DARTS: Differentiable Architecture Search. In *International Conference on Learning Representations*.

[48] Lizheng Ma, Jiaxu Cui, and Bo Yang. 2019. Deep neural architecture search with deep graph bayesian optimization. In *2019 IEEE/WIC/ACM International Conference on Web Intelligence (WI)*. IEEE, 500–507.

[49] Leonel Aguilar Melgar, David Dao, Shaoduo Gan, Nezihe Merve Gürel, Nora Hollenstein, Jiawei Jiang, Bojan Karlas, Thomas Lemmin, Tian Li, Yang Li, Xi Rao,

Johannes Rausch, Cédric Renggli, Luka Rimanic, Maurice Weber, Shuai Zhang, Zhikuan Zhao, Kevin Schawinski, Wentao Wu, and Ce Zhang. 2021. Ease.ML: A Lifecycle Management System for Machine Learning. In *CIDR*.

[50] Supun Nakandala, Arun Kumar, and Yannis Papakonstantinou. 2019. Incremental and approximate inference for faster occlusion-based deep cnn explanations. In *Proceedings of the 2019 International Conference on Management of Data*. 1589–1606.

[51] Supun Nakandala, Yuhao Zhang, and Arun Kumar. 2020. Cerebro: A data system for optimized deep learning model selection. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2159–2173.

[52] Randal S Olson and Jason H Moore. 2019. TPOT: A tree-based pipeline optimization tool for automating machine learning. In *Automated Machine Learning*. Springer, 151–160.

[53] Matthias Poloczek, Jialei Wang, and Peter Frazier. 2017. Multi-information source optimization. In *Advances in Neural Information Processing Systems*. 4288–4298.

[54] Alexander Ratner, Stephen H Bach, Henry Ehrenberg, Jason Fries, Sen Wu, and Christopher Ré. 2020. Snorkel: Rapid training data creation with weak supervision. *The VLDB Journal* 29, 2 (2020), 709–730.

[55] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. 2019. Regularized evolution for image classifier architecture search. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 33. 4780–4789.

[56] Theodoros Rekatsinas, Xu Chu, Ihab F Ilyas, and Christopher Ré. 2017. HoloClean: Holistic Data Repairs with Probabilistic Inference. *Proceedings of the VLDB Endowment* 10, 11 (2017).

[57] Binxin Ru, Xingchen Wan, Xiaowen Dong, and Michael Osborne. 2020. Neural architecture search using bayesian optimisation with weisfeiler-lehman kernel. *arXiv preprint arXiv:2006.07556* 3 (2020).

[58] Rajat Sen, Kirthevasan Kandasamy, and Sanjay Shakkottai. 2018. Noisy Blackbox Optimization with Multi-Fidelity Queries: A Tree Search Approach. *arXiv: Machine Learning* (2018).

[59] Zeyuan Shang, Emanuel Zgraggen, Benedetto Buratti, Ferdinand Kossmann, Philipp Eichmann, Yeounoh Chung, Carsten Binnig, Eli Upfal, and Tim Kraska. 2019. Democratizing data science through interactive curation of ml pipelines. In *Proceedings of the 2019 International Conference on Management of Data*. 1171–1188.

[60] Yu Shen, Yang Li, Jian Zheng, Wentao Zhang, Peng Yao, Jixiang Li, Sen Yang, Ji Liu, and Cui Bin. 2021. ProxyBO: Accelerating Neural Architecture Search via Bayesian Optimization with Zero-cost Proxies. *arXiv preprint arXiv:2110.10423* (2021).

[61] Julien Siems, Lucas Zimmer, Arber Zela, Jovita Lukasik, Margret Keuper, and Frank Hutter. 2020. NAS-Bench-301 and the case for surrogate benchmarks for neural architecture search. *arXiv preprint arXiv:2008.09777* (2020).

[62] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. 2012. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*.

[63] Niranjan Srinivas, Andreas Krause, Sham Kakade, and Matthias Seeger. 2010. Gaussian Process Optimization in the Bandit Setting: No Regret and Experimental Design. In *Proceedings of the 27th International Conference on Machine Learning*. Omnipress.

[64] Louis C. Tiao, Aaron Klein, C. Archambeau, and Matthias W. Seeger. 2020. Model-based Asynchronous Hyperparameter Optimization. *arXiv preprint arXiv:2003.10865* (2020).

[65] Joaquin Vanschoren, Jan N Van Rijn, Bernd Bischl, and Luis Torgo. 2014. OpenML: networked science in machine learning. *ACM SIGKDD Explorations Newsletter* 15, 2 (2014), 49–60.

[66] Manasi Vartak, Harihar Subramanyam, Wei-En Lee, Srinidhi Viswanathan, Saadiyah Husnoo, Samuel Madden, and Matei Zaharia. 2016. ModelDB: a system for machine learning model management. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*. 1–3.

[67] Wei Wang, Jinyang Gao, Meihui Zhang, Sheng Wang, Gang Chen, Teck Khim Ng, Beng Chin Ooi, Jie Shao, and Moaz Reyad. 2018. Rafiki: machine learning as an analytics service system. *Proceedings of the VLDB Endowment* 12, 2 (2018), 128–140.

[68] Ziyu Wang, Masrour Zoghi, Frank Hutter, David Matheson, and Nando De Freitas. 2013. Bayesian optimization in high dimensions via random embeddings. In *Twenty-Third International Joint Conference on Artificial Intelligence*.

[69] Colin White, Willie Neiswanger, and Yash Savani. 2019. Bananas: Bayesian optimization with neural architectures for neural architecture search. *arXiv preprint arXiv:1910.11858* (2019).

[70] Jian Wu, Saul Toscanopalmerin, Peter I Frazier, and Andrew Gordon Wilson. 2019. Practical multi-fidelity Bayesian optimization for hyperparameter tuning. (2019), 284.

[71] Renzhi Wu, Sanya Chaba, Saurabh Sawlani, Xu Chu, and Saravanan Thirumuruganathan. 2020. Zeroer: Entity resolution using zero labeled examples. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1149–1164.

[72] Weiyuan Wu, Lampros Flokas, Eugene Wu, and Jiannan Wang. 2020. Complaint-driven training data debugging for query 2.0. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1317–1334.

[73] Yuhui Xu, Lingxi Xie, Xiaopeng Zhang, Xin Chen, Guo-Jun Qi, Qi Tian, and Hongkai Xiong. 2019. PC-DARTS: Partial Channel Connections for Memory-Efficient Architecture Search. In *International Conference on Learning Representations*.

[74] Quanming Yao, Mengshuo Wang, Yuqiang Chen, Wenyuan Dai, Yu-Feng Li, Wei-Wei Tu, Qiang Yang, and Yang Yu. 2018. Taking human out of learning applications: A survey on automated machine learning. *arXiv preprint arXiv:1810.13306* (2018).

[75] Matei Zaharia, Andrew Chen, Aaron Davidson, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, et al. 2018. Accelerating the machine learning lifecycle with MLflow. *IEEE Data Eng. Bull.* 41, 4 (2018), 39–45.

[76] Xinyi Zhang, Zhuo Chang, Yang Li, Hong Wu, Jian Tan, Feifei Li, and Bin Cui. 2021. Facilitating Database Tuning with Hyper-Parameter Optimization: A Comprehensive Experimental Evaluation. *arXiv preprint arXiv:2110.12654* (2021).

[77] Xinyi Zhang, Hong Wu, Zhuo Chang, Shuowei Jin, Jian Tan, Feifei Li, Tieying Zhang, and Bin Cui. 2021. ResTune: Resource Oriented Tuning Boosted by Meta-Learning for Cloud Databases. In *Proceedings of the 2021 International Conference on Management of Data*. 2102–2114.

[78] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. 2018. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*. 8697–8710.

Multivariate Correlations Discovery in Static and Streaming Data

Koen Minartz
Eindhoven University of Technology
k.minartz@tue.nl

Jens E. d'Hondt
Eindhoven University of Technology
j.e.d.hondt@tue.nl

Odysseas Papapetrou
Eindhoven University of Technology
o.papapetrou@tue.nl

ABSTRACT

Correlation analysis is an invaluable tool in many domains, for better understanding data and extracting salient insights. Most works to date focus on detecting high *pairwise* correlations. A generalization of this problem with known applications but no known efficient solutions involves the discovery of strong multivariate correlations, i.e., finding vectors (typically in the order of 3 to 5 vectors) that exhibit a strong dependence when considered altogether. In this work we propose algorithms for detecting multivariate correlations in static and streaming data. Our algorithms, which rely on novel theoretical results, support two different correlation measures, and allow for additional constraints. Our extensive experimental evaluation examines the properties of our solution and demonstrates that our algorithms outperform the state-of-the-art, typically by an order of magnitude.

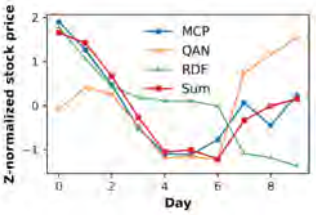
PVLDB Reference Format:
Koen Minartz, Jens E. d'Hondt, and Odysseas Papapetrou. Multivariate Correlations Discovery in Static and Streaming Data. PVLDB, 15(6): 1266-1278, 2022.
doi:10.14778/3514061.3514072

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/CorrelationDetective/public.

1 INTRODUCTION

Correlation analysis is one of the key tools in the arsenal of data analysts for exploring data and extracting insights. For example, in neuroscience, a strong correlation between activity levels in two regions of the brain indicates that these regions are strongly interconnected [11]. In finance, correlation plays a crucial role in finding portfolios of assets that are on the Pareto-optimal frontier of risk and expected returns [16], and in genetics, correlations help scientists detect cause factors for hereditary syndromes.[1] Correlations – as a generalization of functional dependencies – also found use for optimizing access paths in databases [29].

Multivariate correlations, or high-order correlations, are a generalization of pairwise correlations that can capture relations among arbitrarily-sized sets of variables, represented as high-dimensional vectors or as time series.[2] Multivariate correlations have found extensive use in diverse domains: detection of ternary correlations in fMRI time series improved the understanding of how different brain regions work in cohort for executing tasks [1, 2], and in climatology, a ternary correlation led to the characterization of a new weather phenomenon and to improved climate models [15]. Furthermore, a more thorough look at multivariate correlations may open doors in the fields of genomics [23, 30] and medicine [14, 19].



	MCP	QAN	RDF	Sum
MCP	1	0.46	0.53	0.96
QAN	0.46	1	-0.47	0.52
RDF	0.53	-0.47	1	0.52
Sum	0.96	0.52	0.52	1

Figure 1: (a) Normalized daily closing prices for stocks traded at the Australian Securities Exchange, (b) Correlation matrix of the prices.

Accordingly, several measures and algorithms for discovering multivariate correlations have been proposed, such as tripoles [1], multipoles [2], Canonical Correlation Analysis [13] and Total Correlation (TC) [28] and its variants [21, 22, 30]. However, the proposed algorithms do not sufficiently address the fundamental impediment on the discovery of strong multivariate correlations, which is the vast search space. Unfortunately, apriori-like pruning techniques do not apply for the general case of multivariate correlations. For example, consider the three time series presented in Fig. 1, which represent closing prices of three stocks from the Australian securities exchange. In this case, the pairwise correlations between all pairs of the three time series are comparatively low, whereas the time series created by summing QAN and RDF is strongly correlated to MCP. Therefore, a correlation value of any pair of vectors does not provide sufficient information as of whether these vectors may participate together in a ternary (or higher-order) correlation. Simultaneously, an exhaustive algorithm that iterates over all possible combinations implies combinatorial complexity, and cannot scale to reasonably large datasets. Indicatively, in a small data set of 100 vectors, detection of all ternary high correlations requires iterating over 1 million candidates, whereas finding quaternary high correlations among 1000 vectors involves 1 trillion combinations. The mere generation and enumeration of these combinations already becomes challenging. Therefore, smart algorithms are needed that can prune the search space to reduce computational complexity.

Existing algorithms (see Section 2.3) follow at least one of the following approaches: (a) they consider constraining definitions of multivariate correlations that enable apriori-like filtering [2, 21, 30], (b) they rely on hand-crafted additional assumptions of the

Proceedings of the VLDB Endowment, Vol. 15, No. 6 ISSN 2150-8097.
doi:10.14778/3514061.3514072

[1] A prime example is the Spark project for discovering gene properties related to the manifestation of the autism spectrum disorder [9], which led to a list of genes and their correlated symptoms [10].

[2] In the remainder of this paper we will generally refer to the more general case of vectors, but often the data consists of time series that may come with live updates.

user query, which may be too constraining for other application scenarios [1, 2, 30], or, (c) they offer approximate results, with no guarantees [1, 2]. Even though these algorithms are relevant for their particular use cases, they are not generally applicable.

In this work, we follow a more general direction. First, we also consider correlation measures that are unsuitable for apriori-like pruning. Although their usefulness has already been validated in multiple use cases, e.g., [1, 2, 15], algorithms for detecting them do not scale. Second, we consider different algorithmic variants: an exact threshold variant that returns all correlations higher than a threshold τ, and an exact top-κ variant that returns the top-κ highest correlations. We also discuss the case of progressively finding results. Finally, we extend the proposed methods to a dynamic context by efficiently handling streaming data, enabling use-cases where continuous updates of query answers are required, such as flash-trading models in finance [25], weather and server monitoring [27], and neurofeedback training [12, 17, 32].

We evaluate our algorithms on 3 datasets and compare them to the state-of-the-art. Our evaluation demonstrates that we outperform the existing methods by typically an order of magnitude, and the exhaustive-search baseline by several orders of magnitude. Finally, we show that the progressive version of the algorithm produces around 80% of the answers in 10% of the time.

The remainder of the paper is structured as follows. In the next section we formalize the problem, and discuss the preliminaries and related work. We then propose the algorithmic variants for the case of static data (Section 3), and the streaming extension of the algorithm (Section 4). Section 5 summarizes the experimental results. We conclude the paper in Section 6.

2 PRELIMINARIES

We start with a discussion of the multivariate correlation measures that will be considered in this work. We then formalize the problem, and discuss prior work.

2.1 Correlation measures

Our work focuses on two multivariate correlation measures: the two-sided multiple correlation, and the one-sided multipole.

Multiple correlation. Given two sets of vectors X and Y, multiple correlation is defined as follows:

$$\mathbf{mc}(X, Y) = \rho\left(\frac{\sum_{\mathbf{x}\in X}\hat{\mathbf{x}}}{|X|}, \frac{\sum_{\mathbf{y}\in Y}\hat{\mathbf{y}}}{|Y|}\right) \quad (1)$$

where ρ denotes the Pearson correlation coefficient and $\hat{\mathbf{x}}$ denotes $\mathbf{x}$ after z-normalization, i.e., $\hat{\mathbf{x}}_i = \frac{\mathbf{x}_i - \mu_{\mathbf{x}}}{\sigma_{\mathbf{x}}}$. Both the definition and this work can be easily extended to weighted linear aggregates, instead of averaging. Tripoles [1] is a special case of the multiple correlation measure, where $|X| = 2$ and $|Y| = 1$. In this work, we allow both X and Y to contain more vectors.

Multipole. The multipole correlation $\mathbf{mp}(X)$ measures the linear dependence of an input set of vectors X [2]. Specifically, let $\hat{\mathbf{x}}_1, \ldots, \hat{\mathbf{x}}_n$ denote n z-normalized input (column) vectors, and $\mathbf{X} = [\hat{\mathbf{x}}_1, \ldots, \hat{\mathbf{x}}_n]$ the matrix formed by concatenating the vectors. Then:

$$\mathbf{mp}(X) = 1 - \min_{||\mathbf{v}||_2=1} \mathrm{var}(\mathbf{X}\cdot\mathbf{v}) \quad (2)$$

The value of $\mathbf{mp}(X)$ lies between 0 and 1. The measure takes its maximum value when there exists perfect linear dependence, i.e., there exists a vector $\mathbf{v}$ with norm 1, such that $\mathrm{var}(\mathbf{X}\cdot\mathbf{v}) = 0$.

Notice that multipoles is not equivalent to, nor a generalization of, multiple correlation. By definition, **mp** assumes optimal weights (vector $\mathbf{v}$ is such that the variance is minimized), whereas for **mc**, the linear aggregation function for the vectors is determined at the definition of the measure. Furthermore, $\mathbf{mp}(\cdot)$ expresses the degree of linear dependence within a single set of vectors, whereas for $\mathbf{mc}(\cdot,\cdot)$, two distinct, non-overlapping vector sets are considered.

2.2 Problem definition

Consider a set $\mathcal{V} = \{\mathbf{v}_1, \mathbf{v}_2, \ldots \mathbf{v}_n\}$ of d-dimensional vectors, and a multivariate correlation measure *Corr*, both provided by the data analyst. Function *Corr* accepts either one or two vector sets (subsets of $\mathcal{V}$) as input parameters, and returns a scalar. Hereafter, we will be denoting the correlation function as $Corr(X, Y)$, with the understanding that for the definitions of *Corr* that expect one input, Y will be empty. We consider two query types:

Query 1: Threshold query: For a user-chosen correlation function *Corr*, correlation threshold τ, and parameters $l_{\max}, r_{\max} \in \mathbb{N}$, find all pairs of sets $(X \subset \mathcal{V}, Y \subset \mathcal{V})$, for which $Corr(X, Y) \geq \tau$, $X \cap Y = \emptyset$, $|X| \leq l_{\max}$ and $|Y| \leq r_{\max}$.

Query 2: Top-κ query: For a user-chosen correlation function *Corr*, integer parameter κ, and parameters $l_{\max}, r_{\max} \in \mathbb{N}$, find the κ pairs of sets $(X \subset \mathcal{V}, Y \subset \mathcal{V})$ that have the highest values $Corr(X, Y)$, such that $X \cap Y = \emptyset$, $|X| \leq l_{\max}$, and $|Y| \leq r_{\max}$.

The combination of $l_{\max}$ and $r_{\max}$ controls the desired complexity of the answers. Smaller $l_{\max} + r_{\max}$ values yield results that are easier to interpret, and arguably more useful to the data analyst. Complementary to the two query types, users may also want to specify additional constraints, relating to the targeted diversity and significance of the answers. We consider two different constraints, but other constraints (e.g., the weak-correlated feature subset constraint of [30]) can easily be integrated in the algorithm:

Irreducibility constraint: For each (X, Y) in the result set, there exists no (X', Y') in the result set such that $X' \subseteq X$, $Y' \subseteq Y$, and $(X', Y') \neq (X, Y)$. Intuitively, if $Corr(X', Y') \geq \tau$, then no supersets of X' and Y' should be considered together. This constraint prioritizes simpler and more interpretable answers.

Minimum jump constraint: For each (X, Y) in the result set, there exists no (X', Y') such that $X' \subseteq X, Y' \subseteq Y, (X', Y') \neq (X, Y)$, and $Corr(X, Y) - Corr(X', Y') < \delta$. This constraint, which was first proposed in [1], discards solutions where a vector in $X \cup Y$ contributes less than δ to the increase of the correlation.

The minimum jump constraint applies to both query types, whereas the irreducibility constraint is only useful for threshold queries. For top-κ queries, irreducibility is ill-defined: assume $Corr(X, Y) = 0.9$, and $Corr(X', Y') = 0.8$, where $X' \subset X$ and $Y' \subset Y$. In this case, the definition of top-κ does not dictate which of (X, Y) or (X', Y') should be in the answer set.

For conciseness, we will denote the combination of the correlation measure, $l_{\max}$ and $r_{\max}$ as $\mathbf{mc}(l_{\max}, r_{\max})$ (for **mc**) and $\mathbf{mp}(l_{\max})$ (for **mp**). We will call this a *correlation pattern*. For example, $\mathbf{mc}(2, 1)$ will identify the combinations of sets of vectors of size 2 and 1 with high **mc** correlation. Pattern $\mathbf{mp}(4)$ will identify

Table 1: Properties of the most relevant related work for multivariate correlations, and the proposed method.

	Complete	Require constraints	Correlation Measures	Query types
[1]	Yes	Yes	$\mathbf{mc}(1,2)$	Threshold
[2]	No	Yes	$\mathbf{mp}(\cdot)$	Threshold
[21]	No	No	$TC(\cdot)$	Threshold
[30]	Yes	Yes	$TC(\cdot)$ (only binary data)	Threshold
Ours	Yes	No	$\mathbf{mc}(\cdot,\cdot)$, $\mathbf{mp}(\cdot)$	Threshold, top-κ

the combinations of vectors of size at most 4 with high multipoles correlation. Finally, we will denote a particular combination of vectors – a *materialization* of the correlation pattern – by displaying the vectors, grouped by parentheses. For example, $(\mathbf{v}_1, (\mathbf{v}_2, \mathbf{v}_3))$ denotes a combination for the multiple correlation measure, where vectors $\mathbf{v}_2$ and $\mathbf{v}_3$ are aggregated together.

2.3 Related work

Several algorithms exist for efficiently finding highly correlated *pairs* in large sets of high-dimensional vectors, e.g., time series. For example, StatStream [31] and Mueen et al. [20] map pairwise correlations to Euclidean distances, and exploit Discrete Fourier Transforms, grid-based indexing, and dynamic programming to reduce the search space. Other works proposing indices for high dimensional Euclidean data [7, 26] are applicable as well due to the one-to-one mapping of Pearson correlation to Euclidean distance. However, these works are not applicable for multivariate correlations, since two vectors may have a low pairwise correlation with a third vector, whereas their aggregate may have a high correlation (see, e.g., example of Fig. 1).

Agrawal et al. [1] investigate the problem of finding highly-correlated tripoles – a special case of **mc** that contains exactly three vectors. Their algorithm relies on the minimum jump constraint for effective pruning. Compared to tripoles, our work handles the more general definition of multiple correlation, allowing more vectors at the left and right hand side. Moreover, our work does not require the use of the minimum jump constraint to prune comparisons.

Algorithms for discovering high correlations according to the multipole measure (Eqn. 2) were proposed in [2]. Both CoMEt and CoMEtExtended are approximate algorithms relying on clique enumeration and the minimum jump constraint to efficiently explore the search space. Their efficiency depends on a parameter ρ_{CE} that trades off completeness of the result set for performance. Both algorithms yield more complete result sets compared to methods based on l_1-regularization and structure learning. Still, they do not offer completeness guarantees. In contrast, our work is exact – it always retrieves all answers – and outperforms both algorithms.

Total correlation is an non-linear information-theoretic metric that expresses how much information is shared between variables [28]. Nguyen et al. [21] proposed a closely related correlation measure, and an algorithm for finding strongly correlated groups of columns in a database. The key idea of their method is to first evaluate all pairwise correlations, and use those to calculate a lower bound on the total correlation of a group. Their algorithm subsequently finds quasi-cliques in which most pairwise correlations are high, implying a high total correlation value. However, groups with low pairwise correlations can still be strongly correlated as a whole, and these are arguably the most interesting cases. As such, the method is effectively an approximation algorithm. In another work, Zhang et al. also developed an algorithm that discovers sets with a high total correlation value [30]. However, the method is limited to data with binary features, and relies on a limiting weak-correlated subset constraint.

In the supervised learning context, subset regression appears similar to multivariate correlation mining. The goal of this feature selection problem is to select the best p predictors out of n features [6]. Our problem differs from the above in that we aim to find interesting patterns in the data, instead of finding the best predictors for a *given* dependent variable. Further, instead of finding only the single highest correlated set of vectors, our goal is to find a *diverse set* of results, enabling the domain expert to assess a variety of results on qualitative aspects and to gain more insights.

Table 1 summarizes the properties of the most closely related work.

3 DETECTION OF MULTIVARIATE CORRELATIONS IN STATIC DATA

The main challenge in detecting strongly correlated vector sets stems from the combinatorial explosion of the number of combinations that need to be examined. In a dataset of n vectors, there exist at least $O\left(\sum_{p=2}^{l_{\max}+r_{\max}} \binom{n}{p}\right)$ possible combinations. Even if each possible combination can be checked in constant time, their enumeration still requires significant computational effort.

Our algorithm – Correlation Detective, abbreviated as *CD* – exploits the insight that vectors often exhibit (possibly weak) correlations between each other. For example, securities that relate to the same conglomeration (e.g., Fig. 2(a), GOOGL and GOOG) or are exposed to similar risks and opportunities (e.g., STMicroelectronics and ASML) typically exhibit a high correlation between their stock prices. CD exploits such correlations, even if they are weak, to drastically reduce the search space.

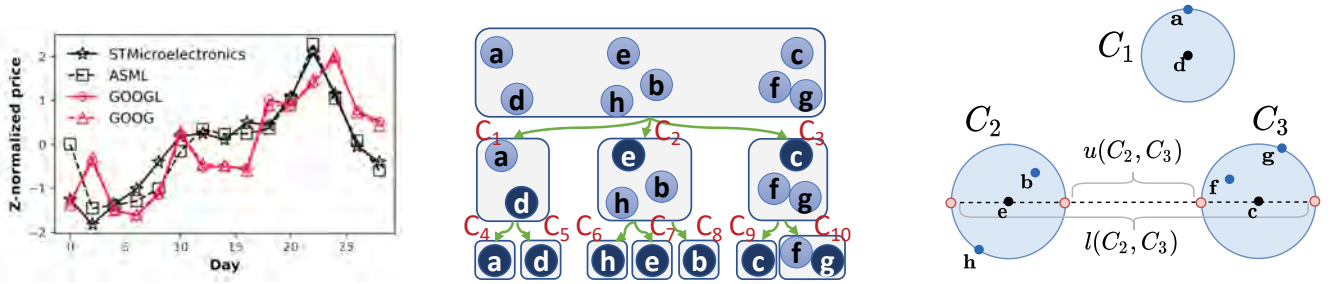
CD works as follows: rather than iterating over all possible vector combinations that correspond to the correlation pattern, CD clusters vectors, and enumerates the combinations of only the cluster centroids. For each of these combinations, it computes an upper and lower bound on the correlations of all vector combinations in the Cartesian product of the clusters. Based on these bounds, CD decides whether or not the combination of clusters (i.e., all combinations of vectors derived from these clusters) should be added to the result set, can safely be discarded, or, finally, if the clusters should be split into smaller subclusters for deriving tighter bounds. This approach effectively reduces the number of combinations that need to be considered.

In the remainder of this section, we will present the algorithm and explain how the two types of queries presented in Section 2 are handled. We will start with a brief description of the initialization and clustering phase. In Sections 3.2 and 3.3 we will describe how CD answers threshold and top-κ queries respectively.

3.1 Initialization and clustering

First, all vectors are z-normalized, i.e., shifted and scaled such that they have zero mean and unit standard deviation. From here on, the algorithm operates only on z-normalized vectors.

Next, we hierarchically cluster all vectors. The clustering algorithm operates in top-down fashion. A root cluster containing all



(a) (b) (c)

Figure 2: (a) Two groups of closely related stocks: ASML and STMicroelectronics are exposed to similar risks, while GOOG and GOOGL participate in the same conglomeration; (b) Running example (schematic): the centroids of each cluster are depicted with darker background. All clusters are labeled for easy reference; (c) Illustration of pessimistic pairwise bounds of Lemma 3.1.

vectors is first created, to initialize the hierarchy. The algorithm then consists of three steps. First, K vectors are picked from the root cluster and used as the initial top-level centroids in the hierarchy. These vectors are picked using the seeding strategy of K-means++ [3]. The use of K-means++ (as opposed to sampling K random vectors) ensures that these initial centroids are well-distributed over the Euclidean space. In the second step, we run standard K-means for at most r_1 iterations, or until convergence, using the average function to recompute the cluster centroids after each iteration. The clustering is evaluated using the Within-Cluster Sum of Squares (WCSS) (the sum of the variances within all clusters). In the third step, steps one and two are repeated r_2 times (i.e., with different initial centroids), and the clustering with the lowest WCSS is kept as the final clustering assignment for the first level of the hierarchy. These three steps are executed recursively on each individual cluster with non-zero radius, to construct the second, third, etc. levels of the hierarchy, until all leaf nodes contain only one vector.

There is a clear tradeoff between the cost of the clustering algorithm and the clustering quality. Increasing the values of r_1 and r_2 results in a higher clustering quality (lower WCSS), but takes longer to compute. However, clustering quality does *not* affect the correctness of CD: regardless of the clustering algorithm, configuration, or final solution, CD always returns the correct results. Poor clustering can only affect the computational efficiency of CD. Still, our experiments show that as long as the clustering is reasonable, a suboptimal clustering is not detrimental to CD's efficiency. More precisely, we found that the value of r_1 (max. iterations of K-means, after the initial centroids were chosen) had no observable effect on CD's efficiency. Therefore, we simply set $r_1 = 1$. The same generally holds for r_2, although to prevent ruinous effects due to coincidentally poorly chosen initial centroids, we set $r_2 = 50$. Still, clustering takes at most a few seconds in our experiments, which is negligible compared to the total execution time of the algorithm.

3.2 Threshold queries

CD receives as input the cluster tree produced by the hierarchical clustering algorithm, a correlation pattern, a correlation function $Corr$, and a correlation threshold τ. It then forms all possible combinations of the correlation pattern with the child clusters of the root. In the example of Fig. 2(b), for a desired correlation pattern of **mc**$(2, 1)$, the following *combinations of clusters* are examined in the order of increasing pattern length:
$\forall_{C_x,C_y \in \{C_1,C_2,C_3\}}(C_x, C_y) \cup \forall_{C_x,C_y,C_z \in \{C_1,C_2,C_3\}}((C_x, C_y), C_z)$

Algorithm 1: ThresholdQuery$(\mathcal{S}_l, \mathcal{S}_r, Corr, \tau)$
Input: Sets of clusters $\mathcal{S}_l$ and $\mathcal{S}_r$ that adhere to the user-defined correlation pattern, correlation measure $Corr$, correlation threshold τ.

```
1  (LB, UB) ← CalcBounds(S_l, S_r, Corr)
2  if LB ≥ τ then
3  |  Add (S_l, S_r) to the result set
4  else if UB < τ then
5  |  Discard (S_l, S_r)
6  else
   |  // Replace largest cluster with subclusters and recurse
7  |  C_max ← arg max_{C ∈ S_l ∪ S_r} {C.radius}
8  |  Set SC ← C_max.subclusters
9  |  for S ∈ SC do
10 |  |  (S'_l, S'_r) ← (S_l, S_r) with C_max replaced by S
11 |  |  ThresholdQuery((S'_l, S'_r), Corr, τ)
```

A combination of clusters compactly represents the combinations created by the Cartesian product of the vectors inside the clusters. For each such combination, the algorithm computes lower and upper bounds on the correlation of these clusters, denoted with LB and UB respectively (Alg. 1, line 1). These bounds, derived later in this section, guarantee that any possible *materialization* of the cluster combination, i.e., replacing each cluster with any one of the vectors in that cluster, will always have a correlation between LB and UB.

The next step is to compare the bounds with the user-chosen threshold τ (lines 2, 4, 6). If $LB \geq \tau$, the combination is *decisive positive*, guaranteeing that all possible materializations of this combination will have a correlation of at least τ. Therefore, all materializations are inserted in the result. If $UB < \tau$, the combination is *decisive negative* – no materialization yields a correlation higher than the threshold τ. Therefore, this combination does not need to be examined further. Finally, when $LB < \tau$ and $UB \geq \tau$, the combination is *indecisive*. In this case, the algorithm (lines 7-11) chooses the cluster $C_{\max}$ with the largest radius, and recursively checks all

combinations where $C_{\max}$ is replaced by one of its sub-clusters. In the example of Figure 2b, assume that the algorithm examined an indecisive combination of clusters C_1, C_2, C_3, and C_2 is the cluster with the largest radius. The algorithm will consider the three children of C_2, and examine their combinations with C_1 and C_3. The recursion continues until each combination is decisive. Decisive combinations are typically found at high levels of the cluster tree, thereby saving many comparisons.

In the following, we will discuss two different approaches for deriving LB and UB for arbitrary correlation patterns. The first approach (theoretical bounds) has constant complexity in the cardinality of the clusters. The second approach (empirical bounds) extends the theoretical bounds with additional information. It has a slightly higher cost, but typically leads to much tighter bounds.

3.2.1 Theoretical bounds. We first present a lemma for bounding the Pearson correlation between only two clusters, which serves as a stepping stone for multivariate correlations.

Lemma 3.1. *Let $\rho(\mathbf{x}, \mathbf{y})$ denote the Pearson correlation between two vectors $\mathbf{x}$ and $\mathbf{y}$, and $\theta_{\mathbf{x},\mathbf{y}}$ the angle formed by these vectors. Consider four z-normalized vectors $\mathbf{u_1}, \mathbf{u_2}, \mathbf{v_1}$, and $\mathbf{v_2}$, such that $\theta_{\mathbf{v_1},\mathbf{u_1}} \leq \theta_1$ and $\theta_{\mathbf{v_2},\mathbf{u_2}} \leq \theta_2$. Then, correlation $\rho(\mathbf{u_1}, \mathbf{u_2})$ can be bounded as follows:*

$$\cos(\theta^{max}_{\mathbf{u_1},\mathbf{u_2}}) \leq \rho(\mathbf{u_1}, \mathbf{u_2}) \leq \cos(\theta^{min}_{\mathbf{u_1},\mathbf{u_2}})$$

where
$\theta^{min}_{\mathbf{u_1},\mathbf{u_2}} = \max\left(0, \theta_{\mathbf{v_1},\mathbf{v_2}} - \theta_1 - \theta_2\right), \theta^{max}_{\mathbf{u_1},\mathbf{u_2}} = \min\left(\pi, \theta_{\mathbf{v_1},\mathbf{v_2}} + \theta_1 + \theta_2\right)$

Proof. All proofs are included in the technical report [18]. □

Lemma 3.1 bounds the correlation between two vectors $\mathbf{u_1}$ and $\mathbf{u_2}$ that belong to two clusters with centroids $\mathbf{v_1}$ and $\mathbf{v_2}$ respectively, by using: (a) the angle between the two centroids, and, (b) upper bounds on the angles between $\mathbf{u_1}$ and $\mathbf{v_1}$, and between $\mathbf{u_2}$ and $\mathbf{v_2}$. For instance, in the running example (Fig. 2(b)), we can bound the correlation between any two vectors from (C_1, C_2) if we have the cosine of the two cluster centroids $\mathbf{d}$ and $\mathbf{e}$, the cosines of $\mathbf{a}$ with $\mathbf{d}$, and $\mathbf{h}$ with $\mathbf{e}$ (as $\mathbf{h}$ is the furthest point in C_2 from the centroid $\mathbf{e}$). The bounds are tightened if the maximum angle formed by each centroid with all cluster vectors is reduced.

We now extend our discussion to cover multivariate correlations, which involve three or more clusters. We first derive bounds for **mc** (Theorem 3.2), and then for **mp** (Theorem 3.3).

Theorem 3.2 (Bounds for **mc**). *For any pair of clusters C_i, C_j, let $l(C_i, C_j)$ and $u(C_i, C_j)$ denote lower/upper bounds on the pairwise correlations between the clusters' materializations, i.e., $l(C_i, C_j) \leq \min_{\mathbf{x}\in C_i, \mathbf{y}\in C_j} \rho(\mathbf{x}, \mathbf{y})$ and $u(C_i, C_j) \geq \max_{\mathbf{x}\in C_i, \mathbf{y}\in C_j} \rho(\mathbf{x}, \mathbf{y})$. Consider the set of clusters $\mathcal{S} = \{C_1, C_2, \ldots, C_N\}$, partitioned into $\mathcal{S}_l = \{C_i\}_{i=1}^{l_{max}}$ and $\mathcal{S}_r = \{C_i\}_{l_{max}+1}^{N}$. Let $L(\mathcal{S}_1, \mathcal{S}_2) = \sum_{C_i\in\mathcal{S}_1, C_j\in\mathcal{S}_2} l(C_i, C_j)$, and $U(\mathcal{S}_1, \mathcal{S}_2) = \sum_{C_i\in\mathcal{S}_1, C_j\in\mathcal{S}_2} u(C_i, C_j)$. Then, for any two sets of vectors $X_l = \{\mathbf{x_1}, \ldots, \mathbf{x_{l_{max}}}\}$, $X_r = \{\mathbf{x_{l_{max}+1}}, \ldots, \mathbf{x_N}\}$ such that $\mathbf{x_i} \in C_i$, multiple correlation $\mathbf{mc}(X_l, X_r)$, can be bounded as follows:*

(1) *if $L(\mathcal{S}_l, \mathcal{S}_r) \geq 0$:*

$$\frac{L(\mathcal{S}_l, \mathcal{S}_r)}{\sqrt{U(\mathcal{S}_l, \mathcal{S}_l)}\sqrt{U(\mathcal{S}_r, \mathcal{S}_r)}} \leq \mathbf{mc}(X_l, X_r) \leq \frac{U(\mathcal{S}_l, \mathcal{S}_r)}{\sqrt{L(\mathcal{S}_l, \mathcal{S}_l)}\sqrt{L(\mathcal{S}_r, \mathcal{S}_r)}}$$

(2) *if $U(\mathcal{S}_l, \mathcal{S}_r) \leq 0$:*

$$\frac{L(\mathcal{S}_l, \mathcal{S}_r)}{\sqrt{L(\mathcal{S}_l, \mathcal{S}_l)}\sqrt{L(\mathcal{S}_r, \mathcal{S}_r)}} \leq \mathbf{mc}(X_l, X_r) \leq \frac{U(\mathcal{S}_l, \mathcal{S}_r)}{\sqrt{U(\mathcal{S}_l, \mathcal{S}_l)}\sqrt{U(\mathcal{S}_r, \mathcal{S}_r)}}$$

(3) *else:*

$$\frac{L(\mathcal{S}_l, \mathcal{S}_r)}{\sqrt{L(\mathcal{S}_l, \mathcal{S}_l)}\sqrt{L(\mathcal{S}_r, \mathcal{S}_r)}} \leq \mathbf{mc}(X_l, X_r) \leq \frac{U(\mathcal{S}_l, \mathcal{S}_r)}{\sqrt{L(\mathcal{S}_l, \mathcal{S}_l)}\sqrt{L(\mathcal{S}_r, \mathcal{S}_r)}}$$

Combined with Lemma 3.1, Theorem 3.2 enables bounding the multiple correlation of any cluster combination that satisfies the correlation pattern, without testing all its possible materializations. For example, for combination $((C_1, C_2), C_3)$ from our running example, we first use Lemma 3.1 to calculate bounds for all cluster pairs in $O(1)$ per pair, which leads to values for $L(\cdot,\cdot)$ and $U(\cdot,\cdot)$. The bounds on $\mathbf{mc}((C_1, C_2), C_3)$ then follow directly from Theorem 3.2.

Also, observe that by tightening the bounds for the pairwise correlations, we can tighten $L(\cdot,\cdot)$ and $U(\cdot,\cdot)$, which will in turn tighten the bounds for **mc**. This is further exploited in Section 3.2.2.

Theorem 3.3 (Bounds for **mp**). *For any pair of clusters C_i, C_j, let $l(C_i, C_j)$ and $u(C_i, C_j)$ denote lower/upper bounds on the pairwise correlations between the cluster's materializations, i.e., $l(C_i, C_j) \leq \min_{\mathbf{x}\in C_i, \mathbf{y}\in C_j} \rho(\mathbf{x}, \mathbf{y})$ and $u(C_i, C_j) \geq \max_{\mathbf{x}\in C_i, \mathbf{y}\in C_j} \rho(\mathbf{x}, \mathbf{y})$. Consider the set of clusters $\mathcal{S} = \left\{C_1, C_2, \ldots, C_{l_{max}}\right\}$. Furthermore, let $\mathbf{L}$ and $\mathbf{U}$ be symmetric matrices with elements $l_{ij} = l(C_i, C_j)$ and $u_{ij} = u(C_i, C_j)$ $\forall 1 \leq i, j \leq l_{max}$. For any set of vectors $X = \{\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_{l_{max}}}\}$ such that $\mathbf{x_i} \in C_i$, multipole correlation $\mathbf{mp}(X)$ can be bounded as follows:*

$$1 - \lambda_{min} - \frac{1}{2}||\mathbf{U} - \mathbf{L}||_2 \leq \mathbf{mp}(X) \leq 1 - \lambda_{min} + \frac{1}{2}||\mathbf{U} - \mathbf{L}||_2$$

where λ_{min} is the smallest eigenvalue of matrix $\frac{\mathbf{L}+\mathbf{U}}{2}$. □

Similar to Theorem 3.2 for **mc**, the tightness of the bounds from Theorem 3.3 depend on the tightness of the bounds for the pairwise correlations between clusters, which can be derived with Lemma 3.1. Proofs for both theorems can be found in [18].

3.2.2 Empirical pairwise bounds. The bounds of Lemma 3.1 – which determine the bounds of Theorems 3.2 and 3.3 – tend to be pessimistic, as they always account for the worst case. In the example of Fig. 2(c), the theoretical lower bound (resp. upper bound) accounts for the case that hypothetical vectors (depicted in pink) are located on the clusters' edges such that they are as far away from (resp. as close to) each other as possible, given the position of the cluster centroids (depicted in black) and cluster radii.

The *empirical bounds* approach builds on the observation that the pairwise correlations of any pair of vectors $\mathbf{x_i}, \mathbf{x_j}$ drawn from a pair of clusters C_i, C_j respectively is typically strongly concentrated around $(l(C_i, C_j) + u(C_i, C_j))/2$, especially for high-dimensional vectors. The approach works as follows. At initialization, we compute all pairwise correlations and store these in an upper-triangular matrix. Note that part of these correlations have already been calculated during the clustering phase. Then, during execution of Alg. 1, we lazily compute $l(C_i, C_j)$ and $u(C_i, C_j)$ as follows: $l(C_i, C_j) = \min_{\mathbf{x}\in C_i, \mathbf{y}\in C_j} \rho(\mathbf{x}, \mathbf{y})$ and $u(C_i, C_j) = \max_{\mathbf{x}\in C_i, \mathbf{y}\in C_j} \rho(\mathbf{x}, \mathbf{y})$, with $\rho(\mathbf{x}, \mathbf{y})$ retrieved from the upper-triangular matrix. The computed $l(C_i, C_j)$ and $u(C_i, C_j)$ are also cached and reused whenever (C_i, C_j)

is encountered in another cluster combination. It is important to note that the empirical bounds do not induce errors, since they trivially satisfy the requirements of Theorems 3.2 and 3.3 that $l(C_i, C_j) \leq \min_{\mathbf{x} \in C_i, \mathbf{y} \in C_j} \rho(\mathbf{x}, \mathbf{y})$ and $u(C_i, C_j) \geq \max_{\mathbf{x} \in C_i, \mathbf{y} \in C_j} \rho(\mathbf{x}, \mathbf{y})$. Consequently, bounds on **mc** and **mp** derived using empirical bounds are still correct. Moreover, they are at least as tight as the bounds of Lemma 3.1, since they account only the vectors that are actually present in the clusters and not the hypothetical worst case.

There is a clear tradeoff between the cost of computing the empirical pairwise bounds (worst case, quadratic to the number of vectors), and the performance improvement of CD from the tighter bounds. Indicatively, in our experiments, the theoretical pairwise bounds computed from Lemma 3.1 were typically between two to eight times wider compared to the empirical pairwise bounds. Exploiting the tighter empirical bounds led to a reduction of the width of the bounds of Theorem 3.2 by 50% to 90% (for $\mathbf{mc}(1, 2)$), which empowered CD to reach to decisive combinations faster. As a result, total execution time of CD with empirical bounds was typically an order of magnitude less than the time with the theoretical bounds. Therefore, all reported results will be using the empirical bounds.

3.2.3 Exploiting additional constraints. CD supports both the irreducibility and minimum jump constraints (see Section 2.2). For irreducibility, the process of identifying whether a simpler combination exists requires testing whether a combination of any of the subsets of $\mathcal{S}_l$ and $\mathcal{S}_r$ is already contained in the answers. To avoid the cost of enumerating all $O(2^{|\mathcal{S}_l|+|\mathcal{S}_r|})$ subsets during the execution of Alg. 1, only the pairwise correlations between any two clusters $C_l \in \mathcal{S}_l$ and $C_r \in \mathcal{S}_r$ are examined (for **mp**, both $C_l \in \mathcal{S}_l$ and $C_r \in \mathcal{S}_l$). Precisely, we use $l(C_l, C_r)$, which is already computed for Theorems 3.2 and 3.3. If there exist C_l, C_r s.t. $l(C_l, C_r) \geq \tau$, then any solution that can be derived from further examining the combination $(\mathcal{S}_l, \mathcal{S}_r)$ cannot satisfy the irreducibility constraint. Therefore, $(\mathcal{S}_l, \mathcal{S}_r)$ can be discarded. The case of minimum jump is analogous: if any $l(C_l, C_r) \geq UB - \delta$, where UB is calculated as in line 1 of Alg. 1, then the combination is discarded. However, considering only the pairwise correlations during the pruning process may lead to inclusion of answers that do not satisfy the constraints. Therefore, such combinations are filtered from the query result before returning it to the user. Since the number of answers is typically in the order of a few tens to thousands, this final pass takes negligible time.

3.3 Top-κ queries

When exploring new datasets, it may be difficult to decide on a threshold τ. Setting the threshold too high for the dataset may lead to no answers, whereas a very low τ can result in millions of answers, and performance decrease. The top-κ variant addresses this issue by allowing users to set the desired number of results, instead of τ. The answer then includes the κ combinations of vectors with the highest correlation that satisfy the correlation pattern.

Assuming an oracle that can predict the τ that would yield κ results, the top-κ queries could be transformed to threshold queries and answered with the standard CD algorithm. Since such an oracle is impossible, many top-κ algorithms (e.g., Fagin's threshold algorithm [8]) start with a low estimate for τ, and progressively increase it, by observing the intermediate answers. The performance of these algorithms depends on how fast they can approach the true value of τ, thereby filtering candidate solutions more effectively.

Algorithm 2: Top-κ-Query($\mathcal{S}_l$, $\mathcal{S}_r$, $Corr$, τ, κ, γ, B)

Input: Sets of clusters $\mathcal{S}_l$ and $\mathcal{S}_r$ that adhere to the user-defined correlation pattern. correlation measure $Corr$, starting threshold τ, desired output set size κ, shrinkfactor γ, list of buckets B.

1 $(LB, UB_{shrunk}) \leftarrow$ CalcBounds($\mathcal{S}_l, \mathcal{S}_r, Corr, \gamma$)
2 **if** $LB \geq \tau$ **then**
3 Add the contents of $(\mathcal{S}_l, \mathcal{S}_r)$ to the result set $\mathcal{R}$
4 $\mathcal{R} \leftarrow$ SORT($\mathcal{R}$)[1:κ]
5 $\tau \leftarrow \min_{(X,Y) \in \mathcal{R}} Corr(X, Y)$
6 **else if** $UB_{shrunk} \geq \tau$ **then**
  // Replace largest cluster with subclusters and recurse with Top-κ-Query (similar to lines 7-11 of Alg. 1)
12 **else**
13 $\gamma^* = \frac{\tau - \mu}{UB - \mu}$
14 Assign $(\mathcal{S}_l, \mathcal{S}_r)$ to bucket $\lceil \gamma^* \cdot |B| \rceil$
  // Phase 2 - starts when Phase 1 is completed
15 **for** $b \in B$ **do**
16 **for** $(\mathcal{S}_l, \mathcal{S}_r) \in b$ **do**
17 ThresholdQuery($\mathcal{S}_l$, $\mathcal{S}_r$, $Corr$, τ)
18 $\mathcal{R} \leftarrow$ SORT($\mathcal{R}$)[1:κ]
19 $\tau \leftarrow \min_{(X,Y) \in \mathcal{R}} Corr(X, Y)$

The top-κ variant of CD (see Alg. 2) follows the same idea. The algorithm has the same core as the threshold-based variant, and relies on two orthogonal techniques to increase τ quickly. First, at invocation, input parameter τ is set to the value of the κ'th highest pairwise correlation. Since all pairwise correlations are computed for the empirical bounds, this causes zero additional cost.

The second technique is an optimistic refinement of the upper bound, aiming to prioritize the combinations with the highest correlations. The algorithm is executed in two phases. In the first phase, similar to Alg. 1, the algorithm computes the upper and lower bound per combination. However, it now artificially tightens the bounds by moving the upper bound towards the lower bound. This so-called *shrinking* is achieved by taking $UB_{\text{shrunk}} = (1-\gamma) \cdot \mu + \gamma \cdot UB$, where $\mu = \frac{UB+LB}{2}$ and $\gamma \in [0, 1]$ is a shrink factor with a default value of 0. If the lower bound surpasses the current threshold τ, all solutions resulting from this candidate combination are added to the set of answers $\mathcal{R}$, and the κ solutions from $\mathcal{R}$ with the highest correlation are kept (Alg. 2, lines 3-4). The value of τ is then set to the minimum correlation in $\mathcal{R}$ (line 5). Otherwise, if UB_{shrunk} is greater than the running τ, we recursively break the cluster to smaller clusters, until we get decisive bounds, analogous to Alg. 1 (lines 6-11). Finally, if the shrunk upper bound is less than the running value of τ but the true UB is greater than τ, we compute the critical shrink factor γ^* for the cluster (line 13) – the minimum value of γ for which UB_{shrunk} would surpass τ. Intuitively, a small γ^* means that the combination is more promising to lead to higher correlation values.

All combinations are placed in B equi-width buckets based on their γ^* values (line 14). At the second phase (lines 15-19), the algorithm processes the buckets one by one, starting from the first, invoking the threshold query algorithm on each of its cluster combinations (Alg. 1) and updating the running τ after every bucket. Since τ continuously increases, and the first buckets are likely to contain the highest correlation values, most combinations after the first few buckets will be filtered without needing many cluster splits.

3.3.1 Progressive threshold queries. The prioritization technique of Alg. 2 can also be used as a basis for a progressive threshold algorithm. Precisely, Alg. 2 can be initialized with a user-chosen τ and with $\kappa \rightarrow \infty$. This will prioritize the combinations that will yield the strongest correlations, and thus also the majority of correlations larger than τ. Prioritization is frequently useful in exploratory data analytics: the user may choose to let the algorithm run until completion, which will yield results identical to Alg. 1, or interrupt the algorithm after receiving sufficient answers. We will evaluate the progressive nature of CD in Section 5.

4 DETECTION OF MULTIVARIATE CORRELATIONS IN STREAMING DATA

Our streaming algorithm, called CDStream, builds on top of CD such that it maintains CD's solution over a sliding window as new data arrive. Currently, CDStream works with the multiple correlation measure only; efficient support for the multipole measure is ongoing work.

CDStream relies on two observations to increase the performance for streaming data. First, most arrivals do not lead to significant updates to the final result. Second, in most real-world scenarios, each of the streams may have a different update rate. For example, in finance, each stock exchange serves updates at different frequencies. The one-size-fits-all approach of CD that handles all updates identically, recomputing the full solution from scratch can be wasteful.

At initialization, CDStream executes CD on the initial data. Then, the core idea of CDStream is as follows. Assume an update of a vector $\mathbf{v}$. This vector belongs to a hierarchy of clusters. For example, vector $\mathbf{e}$ in Fig. 2(b) belongs to C_2 and C_7. We denote the set of these clusters as $C(\mathbf{v})$. The cluster combinations that need to be checked after the update of $\mathbf{v}$ are only the decisive combinations – either positive or negative – that involve a cluster from $C(\mathbf{v})$. The final result remains correct if these combinations are still decisive positive/negative.

To understand how CDStream adds pruning power on top of CD, observe that even for combinations with three or more clusters, the combination's bounds are determined by $l(C_i, C_j)$ and $u(C_i, C_j)$, the minimum and maximum *pairwise* correlations between all involved clusters. Therefore, any update that does not change $l(C_i, C_j)$ and $u(C_i, C_j)$ for all pairs of involved clusters cannot invalidate the previous bounds, or the previous solution. We refer to the pairs of vectors from C_i and C_j that are responsible for $l(C_i, C_j)$ and $u(C_i, C_j)$ as the **minimum** and **maximum extrema pair** respectively. For example, in Fig. 2(c), the minimum and maximum extrema pairs for (C_2, C_3) are $\langle \mathbf{h}, \mathbf{g} \rangle$ and $\langle \mathbf{b}, \mathbf{f} \rangle$ respectively. CDStream exploits this observation by: (a) checking if each update causes a change to any $l(C_i, C_j)$ and $u(C_i, C_j)$, and (b) for the updates that indeed cause a change, updating the extrema pairs, and recomputing the bounds with Theorem 3.2 and the final solution.

Key to the performance of CDStream is an index that enables the algorithm to quickly locate the extrema pairs that are potentially affected by an update. The index maps each vector $\mathbf{v}$ to a list of all decisive combinations (both negative and positive) that involve any cluster from $C(\mathbf{v})$. Internally, the combinations for $\mathbf{v}$ are grouped in two levels. First, they are grouped by the extrema pairs. For example, in Fig. 3, the first 5 combinations for vector $\mathbf{c}$ are grouped under extrema pair $\langle \mathbf{b}, \mathbf{f} \rangle$. All combinations with the same extrema pair are subsequently grouped by the cluster that does not contain the vector used for indexing (in this case, vector $\mathbf{c}$). In our example, the first two combinations have C_2 as the second cluster and are grouped together. We will refer to these clusters for the same extrema pair as the **extrema pair clusters**. The described index is constructed at initialization by iterating over all vectors V in a decisive combination when it is identified (i.e., Alg. 1 lines 3,5), and storing it in the index on every extrema pairs that a vector $\mathbf{v} \in V$ can violate (see Alg. 5 of [18] for pseudocode).

The index is used to support a triggering functionality, which allows us to quickly locate and verify the extrema pairs related to each update. First, the index is used to retrieve the information related to an updated vector $\mathbf{v}$. The algorithm iterates over the respective extrema pairs to verify that these did not change or move, despite the update (Alg. 3, lines 2-11)[3]. Precisely, for each minimum (resp. maximum) extrema pair with correlation $\rho_{\min}$ ($\rho_{\max}$), it verifies that the correlation of $\mathbf{v}$ with all points belonging to the second cluster is still at least $\rho_{\min}$ (at most $\rho_{\max}$) (line 6). If this is still the case, all decisive combinations are still valid. If, on the other hand, an extrema pair is invalidated the respective decisive combinations are checked and their bounds are recomputed and updated. Combinations are re-indexed in case extrema pairs have changed. In case a combination becomes indecisive, subcluster combinations are checked analogously to Alg. 1 lines 7-11, storing new combinations through the standard indexing procedure described earlier. Indecisive combinations are removed from the index in a lazy manner.

Checking whether $\rho_{\min}$ (resp. $\rho_{\max}$) are still valid requires computing the correlation of $\mathbf{v}$ with each of the vectors contained in all extrema pair clusters. A critical observation is that there always exists one cluster in the extrema pair clusters that contains all others – otherwise the other clusters could not contain the same extrema vector. Therefore, the algorithm considers the extrema pair clusters in decreasing size. If the largest cluster passes the test, then all its decisive combinations and all the decisive combinations of all its sub-clusters are still valid and do not need to be checked (lines 9-11). In the running example, if $\langle \mathbf{b}, \mathbf{f} \rangle$ is still the maximum extrema pair between clusters C_3 and C_2, and it has the same $\rho_{\max}$, then all combinations under $\langle \mathbf{b}, \mathbf{f} \rangle$ are still decisive. If the largest cluster does not pass the test, then the bounds for all its decisive combinations are verified. The combinations that are no longer decisive are updated accordingly, e.g., by breaking one of the involved clusters to sub-clusters, as described in Section 3.2. Furthermore, the second,

[3] Alg. 3 describes the process of querying the DCC Index for validating the maximum extrema pairs. The process of validating the minimum extrema pairs is analogous.

Algorithm 3: QueryIndex($i, \mathcal{I}$)

Input: A stream index i, the DCC Index $\mathcal{I}$

Output: A set of DCCs O that need to be checked

```
1  O ← {}                                  // Initialize output set
2  for ⟨a, b⟩ ∈ I[i] do                    // Iterate over extrema pairs
3  |  V ← {}                               // Vectors violating the extrema
4  |  C_p ← I[i][⟨a, b⟩][0]                // Get largest cluster
5  |  for v_j ∈ C_p do                     // Iterate over cluster content
6  |  |  if ρ(v_i, v_j)_t > ρ(a, b)_t then
7  |  |  |  V ← V ∪ v_j                   // Add to violations
8  |  |  |  O ← O ∪ I[i][⟨a, b⟩][C_p]      // Add DCCs
9  |  for C ∈ I[i][⟨a, b⟩][1 :] do         // Check sub-clusters
10 |  |  if C ∩ V ≠ ∅ then                // Violating point in C
11 |  |  |  O ← O ∪ I[i][⟨a, b⟩][C]        // Add DCCs
12 return O
```

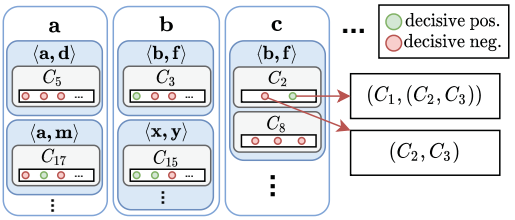


Figure 3: Visualization of the decisive combination index

third, etc. largest extrema pair clusters are tested recursively. The process stops as soon as one of these clusters passes the test.

This grouping of decisive combinations based on the extrema pairs and clusters is instrumental in the algorithm's efficiency, as each pair of clusters may appear in many decisive combinations. In the example of Fig. 2(c), assuming that $l_{\max} + r_{\max} = 3$ with **mc** measure, C_2 and C_3 will appear in a combination of size 2 without C_1, and in a combination of size 3, together with C_1. In both cases, the extrema pairs between C_2 and C_3 will be identical. Therefore, with a single check, both decisive combinations can be verified. Typically the number of decisive combinations for each pair and for each cluster is in the order of a few hundreds for $n = 1000$.

CDStream supports discretization of the stream of updates to small batches (e.g., of a few seconds, or a few tens or hundreds of updates) as a method to trade-off throughput and freshness of results. A larger batch size increases performance and throughput, but potentially delays the updating of the final results. In Section 5 we will evaluate CDStream with different batch sizes.

4.1 User constraints and top-κ queries

To support the minimum jump and irreducibility constraints, additional triggering functionalities, further described below, are added to the index of CDStream.

Irreducibility constraint. Let X, Y, X', Y' denote sets of clusters. Consider combinations (X, Y), and $(X' \subseteq X, Y' \subseteq Y)$, with $|X \cup Y| > |X' \cup Y'|$, i.e., irreducibility excludes (X, Y) from the results if (X', Y') is in. We need to detect two additional cases: (a) (X, Y) needs to be removed from the result set because (X', Y') just surpassed τ, and, (b) (X, Y) needs to be added in the result set, because (X', Y') was just removed from the result set. Both cases can be triggered by an update of a vector from X or Y.

Without the irreducibility constraint, the index contains the following extrema pairs: (a) for the negative decisive combinations, the pairs required for upper-bounding the correlation, (b) for the positive decisive combinations, all pairs required for lower-bounding the correlation. The irreducibility constraint requires also monitoring of the upper bounds of positive decisive combinations (e.g., for case (a), when an increase of $Corr(X', Y')$ will cause the following condition to hold: $Corr(X', Y') > \tau$ which will mean that (X, Y) need to be removed from the result set) and the lower bounds of negative decisive combinations with any $Corr(X', Y') > \tau$. These decisive combinations are also added in the index, under the extrema pairs, and checked accordingly.

Minimum jump constraint. Monitoring for the minimum jump constraint is analogous to the irreducibility contraint. The following cases need to be considered: (a) (X, Y) needs to be removed from the result set because $Corr(X', Y') + \delta > Corr(X, Y)$, and (b) (X, Y) needs to be added in the result set because $Corr(X, Y) > \tau$ and $Corr(X', Y') + \delta < Corr(X, Y)$. Both cases are identified using the discussed method for monitoring the irreducibility constraint.

Top-κ queries Recall that CDStream is initialized with the result of CD. For a top-κ query, CDStream queries CD for a slightly larger number of results $\kappa' = b * \kappa$, where b is at least 1. CDStream finds the minimum correlation in these results, and uses it as a threshold τ in the streaming algorithm. As long as the size of the result set is at least κ, the true top-κ results will always have a correlation higher than τ and will be contained in the top-κ' results maintained by the algorithm. Therefore, the top-κ out of the detected top-κ' correlations are returned to the user.

Scaling factor b controls the tradeoff between the robustness of the streaming algorithm for top-κ queries, and its efficiency. Setting $b = 1$ may lead to the situation that, due to an update, fewer than κ results exist with correlation greater than or equal to τ. CDStream then resorts to CD for computing the correct answer, and updating its index. Conversely, a large b will lead to a larger number of intermediary results, and to more effort for computing the exact correlations of these results, which is necessary for retaining the top-κ results. Our experiments with a variety of datasets have shown that $b = 2$ is already sufficient to provide good performance without compromising the robustness of CDStream.

4.2 CDHybrid: combining CD and CDStream

Recall that CDStream handles the stream updates in batches. The algorithm exhibits high performance when the updates do not drastically change the results set. In streams where the answer changes abruptly, it may be more efficient to run the one-shot algorithm after the completion of each batch and recompute the solution from scratch, instead of maintaining CDStream's index and the result through time. CDHybrid is an algorithm that orchestrates CD and CDStream, transparently managing the switch between the two algorithms based on the properties of the input stream.

To decide between CD and CDStream, CDHybrid needs to estimate the cost of both approaches for handling a batch. A good predictor for this is the number of updates in the batch – more updates tend to cause more changes in the result, which takes longer

for CDStream to handle. Therefore, CDHybrid starts with a brief training period, where it collects statistics on the observed arrival count and execution time of the two algorithms. Simple linear regression is then used to model the relationship between execution time and the observed number of updates. Note that the coefficients of a simple linear regression model can be maintained in constant time and space. Therefore, the regression model is continuously updated, even after the training phase. Switching from one algorithm to the other works as follows.

Switching from CDStream to CD. We cache the current results of CDStream (we will refer to these as $\mathcal{R}_{\text{CDStream}}$) and stop maintaining the index. When a batch is completed, the vectors are updated (i.e., by progressing the sliding window of the each vector) and passed to CD for computing the result.

Switching from CD to CDStream. Since the stream index was not updated for some time, we need to update it before we can use it again. We compute the symmetric difference Δ of the current results of CD (denoted as $\mathcal{R}_{\text{CD}}$) with the last results of CDStream $\mathcal{R}_{\text{CDStream}}$. Any result r contained in $\Delta \cap \mathcal{R}_{\text{CDStream}}$ is due to a negative decisive combination, which needs to be added in the index, whereas any r contained in $\Delta \cap \mathcal{R}_{\text{CD}}$ leads to a new positive decisive combination.

Notice that the switch from CD to CDStream will not remove from the index the decisive combinations that were constructed from CDStream, but are no longer relevant, e.g., because CD split one of its involved clusters. We use a lazy approach to detect these combinations in the index: the first time we access a combination after the switch, we check if there exists a result $r \in \Delta$ that is included in the cluster combination. If so, we reconstruct the combination such that r is removed from it. For example if we access (C_1, C_3), and decisive combination (C_1, C_9) is in Δ, we replace (C_1C_3) with (C_1, C_{10}) and move it to the correct place in the index. If all possible vector combinations in the combination are in Δ, the combination is discarded from the index.

5 EXPERIMENTAL EVALUATION

The purpose of our experiments was twofold: (a) to assess the scalability and efficiency of our methods for varying input parameters, and, (b) to compare them with the state-of-the-art algorithms for multivariate correlation discovery [1, 2], and an exhaustive search baseline that iterates over all possible combinations. The practical significance of multivariate correlations with the two correlation measures was already extensively demonstrated in different domains, e.g., [1, 2, 15] (see Section 1 for more examples). Since CD supports the same correlation measures (and further generalizations of them), and guarantees completeness of results, we do not repeat their use-case studies, but evaluate our methods on the same data (or data of the same type, where the original data was unavailable).

Hardware and implementations. All experiments were executed on a server equipped with a 24-cores Intel Xeon Platinum 8260 Processor, and 400GB RAM. For CoMEtExtended and CONTRa, we used the original implementations, which were kindly provided by the authors [1, 2]. All implementations (including exhaustive search) cached and reused the pairwise correlation computations where applicable, which was always beneficial for performance. The reported execution time for CD and CDStream corresponds to the total execution cost including the steps of pre-processing, clustering and calculating pairwise correlations. All reported results correspond to averages after 10 repetitions.

Datasets. We include results for three real-world datasets.[4] Results with other datasets had similar qualitative outcomes (see [18] for more details).

• **Stocks.** Prices of 1596 stocks, covering a period from April 1, 2020 to May 12, 2020. Each stock has its own update frequency, ranging from 1 to 10 minutes. All prices were normalized with log-return normalization, as is standard in finance. To ensure equal dimensionality, all time series were resampled to 5 minute inter-arrival times for CD (leading to 9103 observations), and missing values were filled with standard interpolation. For CDStream, time series were kept at the original update frequency. We used interpolation to fill missing values, which was required for synchronizing the updates. Notice that any algorithm could be used instead for this process, e.g., forward or backward-filling, or even a more complex solution that incorporates ML, e.g., a deep neural network [4].

• **fMRI.** Functional MRI data of a participant watching a movie, prepared with the recommended steps for voxel-based analytics. The data was further pre-processed by mean-pooling with kernels of 2x2x2, 3x3x3, 4x4x4, 6x6x6 and 8x8x8 voxels, each representing the mean activity level at a cube of voxels in the scan. Subsequently, constant-value time series were removed. This led to a total of 9700, 3152, 1440, 509 and 237 time series respectively, all of equal length (5470 observations), covering a period of ~1.5 hours. Unless otherwise mentioned, the reported results correspond to the 4x4x4 resolution, i.e. 1440 time series.

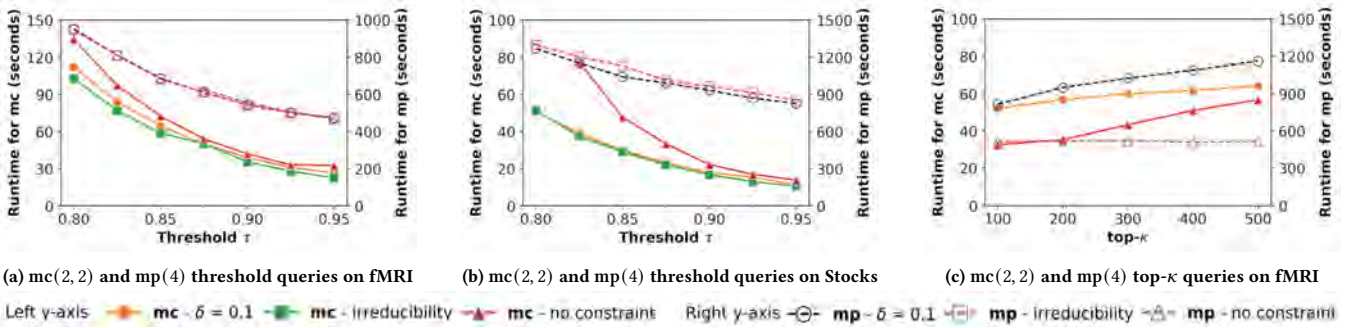
• **SLP.** Sea Level Pressure data [24], as preprocessed in [2]. The dataset contains 171 time series, each with 108 observations.

5.1 CD on static data

5.1.1 Threshold queries. Figs. 4a-b show the effect of threshold τ on execution time of CD for the fMRI and Stocks dataset respectively. The left Y axis corresponds to query $\mathbf{mc}(2, 2)$ with different constraints, whereas the right Y axis corresponds to $\mathbf{mp}(4)$. The plot does not include a result for **mc** in the Stocks dataset for $\tau = 0.8$, since the query returned more than 10 Million results, and our implementation automatically switches to the top-κ variant (with $\kappa = 10^7$) in such cases. Our first observation is that increasing the threshold consistently leads to higher efficiency. This is expected, since a higher threshold enables more aggressive pruning of candidate comparisons. Furthermore, CD is noticeably faster for **mc** compared to **mp**. This is due to two reasons: (a) the complexity of the computation of eigenvalues of a matrix (cubic to $l_{\max}$), which is required for computing the bounds for **mp** (Theorem 3.3), and (b) **mp** typically results in higher correlation values and to more answers for the same value of τ compared to **mc**.

We found that the individual execution times over the 10 repetitions for each configuration were stable, with a relative standard deviation typically between 1%-2%, or below 5 seconds in absolute value. The maximum relative standard deviation for a configuration observed in all experiments was 4.7% of the mean query time.

[4] See https://github.com/CorrelationDetective/public for download links, instructions, and code for reading the data.



(a) mc$(2,2)$ and mp(4) threshold queries on fMRI **(b) mc$(2,2)$ and mp(4) threshold queries on Stocks** **(c) mc$(2,2)$ and mp(4) top-κ queries on fMRI**

Figure 4: Effect of constraints, τ and κ on performance.

5.1.2 Top-κ queries. Fig. 4c shows the execution time of CD for different values of κ with the fMRI dataset – the results with Stocks were qualitatively very similar. We see that a decrease of κ typically leads to increased efficiency. A low value of κ helps the algorithm to increase the running threshold τ faster, leading to more aggressive pruning when Alg. 1 is invoked. Interestingly, this behavior is not as prevalent for **mp**(4) with no constraints. This discrepancy can be attributed to the correlation values in the result set. Indicatively, for this query the lowest correlation in the result set only decreases from 0.917 (top-100) to 0.915 (top-500). In contrast, the same pattern with a minimum jump constraint of $\delta = 0.1$ shows a decrease in this correlation from 0.82 (top-100) to 0.78 (top-500), explaining why the effect of κ on performance is more substantial.

5.1.3 Correlation pattern. Table 2 presents the results size and execution time of CD for different correlation patterns. As expected, increasing the complexity of the correlation pattern leads to an increase of the computational time. However, even though the size of the search space follows $O\left(\binom{n}{l_{\max}+r_{\max}}\right)$, execution time of CD grows at a much slower rate. Indicatively, for the fMRI dataset, the search space size grows 5 orders of magnitude between **mc**$(1,2)$ and **mc**$(1,4)$. Execution time increases by only three orders of magnitude, indicating efficient pruning of the search space.

5.1.4 Clustering sensitivity. We now analyze the sensitivity of CD with respect to the hierarchical clustering parameters. Since correctness of CD is not influenced by the clustering, our experiments only investigate its influence on the efficiency of CD. Table 3 illustrates the effect of K (the number of sub-clusters per cluster) on CD's execution time. A very small number of sub-clusters in each split ($K = 2$) hurts efficiency significantly, as it results in extremely large clusters at the high levels of the hierarchy, and Algorithm 1 needs to drill deeper into the hierarchy before reaching to decisive combinations. Very high K values also lead to suboptimal performance. In that case, the clusters are more compact, leading to decisive combinations at higher levels, but more cluster combinations exist (in these higher levels) that need to be considered.

For K around 10, CD's efficiency is reasonably robust. In fact, setting $K = 10$ led to performance close to the optimal in all cases – at most 15% worse than the optimal performance for the same query, or at most 5% if we do not consider absolute differences up to 10 seconds. The small impact of K, as long as it is not close to the extremes, can be explained by considering how it affects the depth of the clustering tree: intuitively, under the simplifying assumption that each cluster contains approximately an equal amount of vectors, the depth of the clustering hierarchy is approximately $\log_K(n)$. This depth does not vary significantly with the value of K. Indicatively, for 1000 vectors, setting $K \in [10, 30]$ leads to a hierarchy of 3 to 4 levels. Therefore, as long as we avoid extremely small and extremely large K values, the impact of K to CD's efficiency is small. For consistency, for all our remaining experiments we set $K = 10$.

5.1.5 Progressive variant. We also evaluated the progressive nature of CD. We modified our code such that it tracks the number of discovered results at different time points. Figure 5b plots the number of results returned by the algorithm on the Stocks dataset, as a function of time. The results correspond to correlation patterns **mc**$(1,4)$ and **mp**(4), which take significant time to complete, since these are the ones that would mostly benefit from a progressive algorithm. We see that CD retrieves around half of the results in the first few seconds, and already reaches 80% recall in around 10% of the total execution time.

5.1.6 Comparison to exhaustive search baseline. Figure 5a plots the execution time of CD and the exhaustive baseline for processing fMRI datasets of different sizes, obtained as discussed in Section 5. The results correspond to correlation patterns **mc**$(1,3)$ and **mp**(3). We see that execution time of CD for both patterns grows at a slower rate compared to the exhaustive search method, and the difference increases with the dataset size. This finding is consistent with our earlier observation that the runtime of CD grows slower than the size of the search space (Section 5.1.3), meaning that CD can handle significantly larger datasets than the exhaustive algorithm in reasonable time.

5.1.7 Comparison to CoMEtExtended. Our next experiment focused on comparing CD with CoMEtExtended [2]. The goal of CoMEtExtended differs slightly from our problem statement. First, CoMEtExtended is approximate. Even though it does not offer approximation guarantees, its recall (and efficiency) can be tuned by parameter ρ_{CE}, which takes values between -1 and 1. Values around 0 offer a reasonable tradeoff between efficiency and recall; when CoMEtExtended is configured to return the exact result set ($\rho_{\text{CE}} = 1$), it degenerates to exhaustive search [2], to which we compared in Section 5.1.6. In contrast, CD always produces complete answers. Therefore, we consider both execution time and recall rate in our comparison. Second, CoMEtExtended aims to find only *maximal* sets that exhibit a strong **mp** correlation, whereas CD finds *all* sets (up to a specified cardinality) that are strongly correlated. To

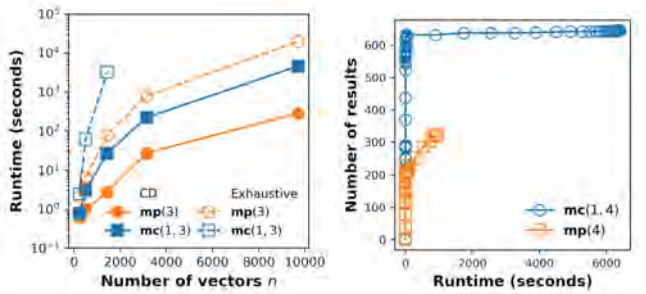


Figure 5: (a) Running time of CD (filled markers) and exhaustive algorithm (empty markers, dashed lines) for varying resolutions of the fMRI dataset, with $\tau = 0.9$ and no constraints. Queries were interrupted after 20 hours. (b) Number of retrieved results in relation to runtime, for progressive execution of $\mathbf{mc}(1, 4)$ and $\mathbf{mp}(4)$, with $\tau = 0.9, \delta = 0.05$ on the Stocks dataset.

ensure a fair comparison for CoMEtExtended, we also considered all subsets of each result returned by CoMEtExtended. When a subset of a CoMEtExtended answer satisfied the query, we added it to the results, thereby increasing CoMEtExtended's recall. This step was not included in the execution time of CoMETExtended, i.e., it did not penalize its performance. Conversely, instead of enhancing the results of CoMETExtended we could filter out the non-maximal results from CD's result set. Since both approaches led to a very similar comparison (recall and execution time), we present only the results of the first approach. Table 5 presents the number of results and execution time of CoMEtExtended and CD on the same dataset (SLP) and parameters used in [2]. We only consider the **mp** measure, since CoMEtExtended does not support **mc**. We see that CD is consistently faster than CoMEtExtended – at least an order of magnitude – and often returns substantially more results. Indicatively, for $\mathbf{mp}(4)$, CoMEtExtended with $\rho_{\text{CE}} = 0$ (resp. $\rho_{\text{CE}} = 0.02$) is one to two (resp. two to three) orders of magnitude slower than CD. Notice that for queries with $\delta = 0.1$, CoMEtExtended found 281 results with 6 vectors, and one with 7 ($\rho_{\text{CE}} = 0.02, \tau = 0.4$). These amount to $\sim 0.3\%$ of the total amount of discovered results. These were not discovered by CD, as the queries specified $l_{\max} = 5$ at most, prioritizing the simpler and more interpretable results. Nevertheless, for these settings, CD still found 25% more results than COMEtExtended, and in one fourth of the time. Moreover, the case studies presented in [1, 2], amongst others on this dataset, demonstrate the usefulness and significance of relatively simple relationships, involving at most four time series. Other works on multivariate correlations also emphasize the discovery of relationships that do not contain too many time series [5]. For these cases, with a fixed $l_{\max}$, CD is guaranteed to find a superset of COMEtExtended's result set, at a fraction of the time.

5.1.8 Comparison to CONTRa. We also compared CD to CONTRa [1] for discovery of tripoles, i.e., $\mathbf{mc}(1, 2)$ correlations. For a fair comparison, CD was parameterized to find the same results as CONTRa and to utilize the same hardware, as follows: (a) CD was executed with $\tau = 0$, i.e., pruning was solely due the minimum jump constraint, and (b) CD was configured to utilize only one thread/core, since the implementation of CONTRa was single-threaded. CONTRa was configured to return the exact results.

The experimental results with the fMRI dataset are shown in Table 6.[5] We see that CD is more efficient than CONTRa for detecting the same results, even with $\tau = 0$. However, the lack of τ yields an impractically large amount of results. As such, we also evaluate CD with $\tau = 0.5$ (corresponding to the lowest correlation reported in the case studies of [1]) and $\tau = 0.9$ (which gives a reasonable amount of results, in the order of a few tens to hundreds). This further decreases the runtime of CD by one to two orders of magnitude, while preventing clutter of the result set by returning only the most strongly correlated triplets.

5.2 Evaluation with streaming data

The second set of experiments was configured to evaluate the performance of CDStream. We used the timestamps that are contained in the three datasets for ordering the data and generating the streams. Our discussion will focus on the Stocks dataset; results for the other datasets are shown only when they offer new insights. Unless otherwise mentioned, the following results correspond to a batch size of 50, a sliding window of 2000, and a dataset size of 1000 stocks.

5.2.1 Comparison to CD. Fig. 6a presents CDStream's mean processing time per batch, for different dataset sizes created by randomly picked stocks. The figure also includes the average time required for executing CD at the end of each batch. We see that CDStream is more efficient than CD for small correlation patterns, requiring a few milliseconds. Note that, even though the number of comparisons increases at a combinatorial rate with the number of vectors , the execution time of CDStream grows substantially slower. This is due to the grouping technique in the index of CDStream, which effectively reduces the work for processing each update. For more complex patterns, e.g., $\mathbf{mc}(2, 3)$, CDStream has performance comparable to CD.

5.2.2 Effect of query parameters. Table 4 presents the effect of τ and constraints (minimum jump and irreducibility) on CDStream's performance. We see that efficiency of CDStream is robust to constraints – a constraint only causes a small difference in the number of decisive combinations that need to be monitored. In contrast, an increasing value of τ leads to better performance, as decisive combinations are reached earlier, similar to the case of CD.

Figure 6b plots the average processing time per update, for varying batch sizes and for both fMRI and Stocks. The batch size (X-axis) is presented as a multiplicative factor on the number of vectors n in each dataset. We see that the batch size enables tuning the tradeoff between throughput and update rate of the results: increasing the batch size increases efficiency, but reduces freshness of results. This happens because both algorithms will process only the latest values for each vector, ignoring intermediary updates. Also observe that CD's efficiency approaches that of CDStream as the batch size increases. For Stocks, processing time for the two algorithms crosses at a batch size $4 * n$, whereas for fMRI, this crossing happens at batch size $8 * n$. This discrepancy can be attributed to the properties of the datasets (the inherent distributions and magnitude of updates) and exemplifies the importance of CDHybrid. As we will see shortly (Section 5.2.4), CDHybrid is able to adapt to the

[5] For this experiment, the minimum jump parameter δ is defined as in [1], to represent the minimum difference between the *squared* correlations.

Table 2: CD with different correlation patterns.

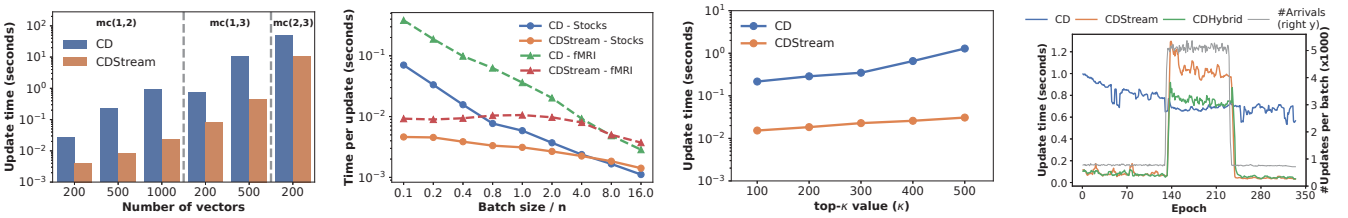
	fMRI		Stocks	
	time (s)	#results	time (s)	#results
mc$(1,2)$	1.4	53	2.0	581
mc$(1,3)$	26.8	1350	23.3	632
mc$(2,2)$	41.5	4239	18.2	1875
mc$(1,4)$	6294	42196	6369.9	646
mc$(2,3)$	15760	287651	4238.6	2796
mp(3)	2.6	33	4.6	302
mp(4)	560.0	58213	966.4	576

Table 3: Execution times (in seconds) for varying clustering parameters and queries ($\delta = 0.05$).

	$\tau \backslash K$	fMRI 2	5	10	25	50	Stocks 2	5	10	25	50
mc$(1,3)$	0.8	446	**108**	121	131	157	722	106	106	142	**104**
	0.9	140	**35**	40	49	63	281	**23**	27	48	36
mc$(2,2)$	0.8	883	**174**	188	177	197	715	78	**75**	92	80
	0.9	264	**57**	64	68	94	179	**22**	**22**	35	30
mp(4)	0.8	4799	1037	**1014**	1061	1149	10547	**1366**	1369	1809	1424
	0.9	2451	**592**	606	641	706	6808	1020	**981**	1497	1200

Table 4: Effect of τ and δ on CD and CDStream for streaming data, with Stocks.

	CD			CDStream		
$\delta \backslash \tau$	**0.6**	**0.7**	**0.8**	**0.6**	**0.7**	**0.8**
None	3.12	2.62	0.80	.045	.036	.023
Irred.	3.21	3.26	0.88	.046	.036	.023
0.05	3.87	2.39	0.93	.043	.034	.023
0.10	3.00	2.77	1.13	.044	.033	.023
0.15	3.15	2.49	1.14	.043	.033	.022



(a) Effect of dataset size and correlation pattern, with $\delta = 0.05$, $\tau = 0.8$, Stocks.

(b) Effect of batch size, $\delta = 0.05$, $\tau = 0.8$.

(c) Effect of κ, with $\delta = 0.05$, with Stocks.

(d) Efficiency of CDHybrid over time, with Stocks.

Figure 6: Effect of query parameters on performance of CDStream.

Table 5: Comparison of CoMEtExtended and Correlation Detective on SLP: running time (seconds) and number of retrieved results.

τ, δ	CoMEtExtended $\rho_{CE} = 0$ time	#res.	$\rho_{CE} = 0.01$ time	#res.	$\rho_{CE} = 0.02$ time	#res.	Correlation Detective **mp**(4) time	#res.	**mp**(5) time	#res.
0.4, 0.1	604	62663	1318	67110	3530	70921	11	71083	899	88305
0.4, 0.15	511	7244	1218	7300	3393	7343	9	7559	575	7562
0.4, 0.2	501	2166	1210	2171	3327	2174	7	2183	333	2183
0.5, 0.1	459	30632	1099	33718	2836	36457	7	34592	557	51391
0.5, 0.15	398	3646	1006	3702	2760	3745	6	3961	391	3964
0.5, 0.2	390	1434	1006	1439	2701	1442	6	1451	292	1451
0.6, 0.1	246	7823	598	8892	1592	9859	5	9204	310	17349
0.6, 0.15	223	1569	577	1606	1559	1635	5	1840	245	1843
0.6, 0.2	219	771	568	776	1532	779	5	788	199	788

Table 6: Comparison of CONTRa and CD: running time (seconds) and number of results for the largest fMRI dataset ($n = 9700$).

δ	CONTRa time	results	CD ($\tau = 0$) time	results	CD ($\tau = 0.5$) time	results	CD ($\tau = 0.9$) time	results
0.1	>24hrs	22952036	17027	22952036	3602	20527560	458	432
0.15	11162	733018	7168	733018	3151	732908	458	102
0.2	5324	20555	3852	20555	2790	20555	459	24

properties of the dataset, and chooses the best algorithm. In [18] we also report on the sensitivity of CDStream to the sliding window size, and consider more values for batch size.

5.2.3 Top-κ queries. Fig. 6c plots the average processing time per batch for top-κ query **mc**$(1,2)$, for different κ values. We see that processing time for both algorithms increases with κ. In CD, execution time grows almost linearly with κ (from 200 msec to almost 1.3 second), whereas for CDStream the time increases by only a factor of two for the same values. The reason for this notable difference in efficiency is that CDStream only maintains the top-κ solutions, already having a good estimate for the threshold of the top-κ highest correlation from previous runs, whereas CD has to start each run from scratch to avoid finding less than κ results.

5.2.4 CDHybrid. For this experiment, we use a time-based batch size of 1 minute, and simulate stream bursts by speeding up the updates (reducing the inter-arrival times) around the middle of the stream, for approximately one third of the stream length. Figure 6d depicts the processing time per batch (moving average for the last 5 batches), for processing Stocks with CD, CDStream, and CDHybrid. Each epoch corresponds to one batch. The figure also includes the number of arrivals within each batch (right Y axis). We observe that CDHybrid quickly switches to the best method. Shortly after a switch from CDStream to CD, the cost of CDHybrid is slightly higher compared to the optimal cost. This is attributed to the initialization cost of CD. For the case of switching back to CDStream (epoch 240), the additional cost for updating the outdated index is also small, indicating that the process of updating the index after the switch is not expensive. Also recall that part of this cost (for removing the expired decisive combinations from the index) is amortized through a large number of epochs, due to the lazy updating algorithm discussed in Section 4.2. Particularly, switching to CDStream has a cumulative cost of 0.109 seconds, amortized over 60 epochs, amounting to $\sim 3.8\%$ of the total processing time over these epochs. The cost of CDHybrid to decide between the two algorithms was negligible in all cases, requiring less than 0.1 msec. This cost is already included in the shown results.

6 CONCLUSIONS

We considered the problem of detecting high multivariate correlations with two correlation measures, and with different constraints. We proposed three algorithms: (a) CD, optimized for static data, (b) CDStream, which focuses on streaming data, and (c) CDHybrid for streaming data, which autonomously chooses between the two algorithms. The algorithms rely on novel theoretical results, which enable us to bound multivariate correlations between large sets of vectors. A thorough experimental evaluation using real-world datasets showed that our contribution outperforms the state of the art typically by an order of magnitude.

REFERENCES

[1] Saurabh Agrawal, Gowtham Atluri, Anuj Karpatne, William Haltom, Stefan Liess, Snigdhansu Chatterjee, and Vipin Kumar. 2017. Tripoles: A New Class of Relationships in Time Series Data. In *Proceedings of the 23rd SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 697–706.

[2] Saurabh Agrawal, Michael Steinbach, Daniel Boley, Snigdhansu Chatterjee, Gowtham Atluri, Anh The Dang, Stefan Liess, and Vipin Kumar. 2020. Mining Novel Multivariate Relationships in Time Series Data Using Correlation Networks. *IEEE TKDE* 32, 9 (2020), 1798–1811.

[3] David Arthur and Sergei Vassilvitskii. 2007. K-Means++: the advantages of careful seeding. In *Proc. 18th Annual Symposium on Discrete Algorithms, SODA*, Nikhil Bansal, Kirk Pruhs, and Clifford Stein (Eds.). SIAM, 1027–1035.

[4] Wei Cao, Dong Wang, Jian Li, Hao Zhou Bytedance, A I Lab, Yitan Li, Bytedance Ai Lab, and Lei Li. 2018. BRITS: Bidirectional Recurrent Imputation for Time Series. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems* (Montréal, Canada). 6776–6786.

[5] Roger H.L. Chiang, Chua Eng Huang Cecil, and Ee-Peng Lim. 2005. Linear correlation discovery in databases: a data mining approach. *Data & Knowledge Engineering* 53, 3 (2005), 311–337.

[6] Abhimanyu Das and David Kempe. 2008. Algorithms for Subset Selection in Linear Regression. In *Proc. 40th ACM Symposium on Theory of Computing (STOC '08)*. ACM, 45–54.

[7] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. 2004. Locality-Sensitive Hashing Scheme Based on p-Stable Distributions. In *Proc. 20th Annual Symposium on Computational Geometry (SCG '04)*. ACM, 253–262.

[8] Ronald Fagin, Amnon Lotem, and Moni Naor. 2001. Optimal Aggregation Algorithms for Middleware. *J. Comput. System Sci.* 66 (2001), 614–656.

[9] Simons Foundation. 2021. SPARK for Autism. https://sparkforautism.org/portal/page/autism-research/. Accessed: 2021-07-30.

[10] Simons Foundation. 2021. SPARK Gene list. https://d2dxtcm9g2oro2.cloudfront.net/wp-content/uploads/2020/07/13153839/SPARK_gene_list_July2020.pdf. Accessed: 2021-07-30.

[11] Daniel A. Handwerker, Vinai Roopchansingh, Javier Gonzalez-Castillo, and Peter A. Bandettini. 2012. Periodic changes in fMRI connectivity. *NeuroImage* 63, 3 (2012), 1712–1719.

[12] Stephan Heunis, Rolf Lamerichs, Svitlana Zinger, Cesar Caballero-Gaudes, Jacobus F.A. Jansen, Bert Aldenkamp, and Marcel Breeuwer. 2020. Quality and denoising in real-time functional magnetic resonance imaging neurofeedback: A methods review. *Human Brain Mapping* 41, 12 (2020), 3439–3467.

[13] Wolfgang Karl Härdle. 2007. *Applied Multivariate Statistical Analysis* (2 ed.). Springer. 321–330 pages.

[14] Silvan Licher, Shahzad Ahmad, Hata Karamujić-Čomić, Trudy Voortman, Maarten J. G. Leening, M. Arfan Ikram, and M. Kamran Ikram. 2019. Genetic predisposition, modifiable-risk-factor profile and long-term dementia risk in the general population. *Nature Medicine* 25, 9 (2019), 1364–1369.

[15] Stefan Liess, Saurabh Agrawal, Snigdhansu Chatterjee, and Vipin Kumar. 2017. A Teleconnection between the West Siberian Plain and the ENSO Region. *Journal of Climate* 30, 1 (2017), 301 – 315.

[16] Myles E. Mangram. 2013. A Simplified Perspective of the Markowitz Portfolio Theory. *Global Journal of Business Research* 7, 1 (2013), 59–70.

[17] Fukuda Megumi, Ayumu Yamashita, Mitsuo Kawato, and Hiroshi Imamizu. 2015. Functional MRI neurofeedback training on connectivity between two regions induces long-lasting changes in intrinsic functional network. *Frontiers in Human Neuroscience* 9 (2015).

[18] Koen Minartz, Jens d'Hondt, and Odysseas Papapetrou. 2021. *Multivariate correlation discovery in static and streaming data.* Technical Report. Eindhoven University of Technology. Available in https://github.com/CorrelationDetective/public.

[19] Ileena Mitra, Alinoë Lavillaureix, Erika Yeh, Michela Traglia, Kathryn Tsang, Carrie E. Bearden, Katherine A. Rauen, and Lauren A. Weiss. 2017. Reverse Pathway Genetic Approach Identifies Epistasis in Autism Spectrum Disorders. *PLOS Genetics* 13, 1 (01 2017), 1–27. https://doi.org/10.1371/journal.pgen.1006516

[20] Abdullah Mueen, Suman Nath, and Jie Liu. 2010. Fast Approximate Correlation for Massive Time-Series Data. In *Proc. ACM International Conference on Management of Data (SIGMOD '10)*. ACM, 171–182.

[21] Hoang Vu Nguyen, Emmanuel Müller, Periklis Andritsos, and Klemens Böhm. 2014. Detecting Correlated Columns in Relational Databases with Mixed Data Types. In *Proc. 26th International Conference on Scientific and Statistical Database Management (SSDBM '14)*. ACM, Article 30, 12 pages.

[22] Hoang Vu Nguyen, Emmanuel Müller, Jilles Vreeken, Pavel Efros, and Klemens Böhm. 2014. Multivariate Maximal Correlation Analysis. In *Proc. 31st International Conference on Machine Learning - Volume 32 (ICML'14)*. 775–783.

[23] Örjan Carlborg and Chris S. Haley. 2004. Epistasis: too often neglected in complex trait studies? *Nature Reviews Genetics* 5, 8 (2004), 618–625.

[24] Kistler RE, Eugenia Kalnay, William Collins, Suranjana Saha, G. White, John Woollen, Muthuvel Chelliah, Wesley Ebisuzaki, Masao Kanamitsu, Vernon Kousky, Huug Dool, Jenne RL, and Mike Fiorino. 2001. The NCEP/NCAR 50-year reanalysis: monthly means CD-ROM and documentation. *Bulletin of the American Meteorological Society* 82 (2001), 247–268.

[25] Camilo Rostoker, Alan Wagner, and Holger Hoos. 2007. A Parallel Workflow for Real-time Correlation and Clustering of High-Frequency Stock Market Data. In *Proc. 21th International Parallel and Distributed Processing Symposium*. 1–10.

[26] Venu Satuluri and Srinivasan Parthasarathy. 2012. Bayesian Locality Sensitive Hashing for Fast Similarity Search. *Proc. VLDB Endow.* 5, 5 (2012), 430–441.

[27] Zhiyuan Tan, Aruna Jamdagni, Xiangjian He, Priyadarsi Nanda, and Ren Ping Liu. 2014. A system for denial-of-service attack detection based on multivariate correlation analysis. *Trans. Parallel and Distributed Systems (TPDS)* 25, 2 (2014), 447–456.

[28] Satosi Watanabe. 1960. Information Theoretical Analysis of Multivariate Correlation. *IBM Journal of Research and Development* 4, 1 (1960), 66–82.

[29] Yingjun Wu, Jia Yu, Yuanyuan Tian, Richard Sidle, and Ronald Barber. 2019. Designing Succinct Secondary Indexing Mechanism by Exploiting Column Correlations. In *Proc. International Conference on Management of Data (SIGMOD'19)*. ACM, 1223–1240.

[30] Xiang Zhang, Feng Pan, Wei Wang, and Andrew Nobel. 2008. Mining non-redundant high order correlations in binary data. *Proc. VLDB Endow.* 1, 1 (2008), 1178–1188.

[31] Yunyue Zhu and Dennis Shasha. 2002. StatStream: Statistical Monitoring of Thousands of Data Streams in Real Time. In *Proc. 28th International Conference on Very Large Data Bases (VLDB '02)*. 358–369.

[32] Anna Zilverstand, Bettina Sorger, Jan Zimmermann, Amanda Kaas, and Rainer Goebel. 2014. Windowed Correlation: A Suitable Tool for Providing Dynamic fMRI-Based Functional Connectivity Neurofeedback on Task Difficulty. *PLOS ONE* 9, 1 (01 2014), 1–13.

Moneyball: Proactive Auto-Scaling in Microsoft Azure SQL Database Serverless

Olga Poppe, Qun Guo, Willis Lang, Pankaj Arora, Morgan Oslake, Shize Xu, and Ajay Kalhan
Microsoft Corporation, One Microsoft Way, Redmond, WA 98052, USA
firstname.lastname@microsoft.com

ABSTRACT

Microsoft Azure SQL Database is among the leading relational database service providers in the cloud. Serverless compute automatically scales resources based on workload demand. When a database becomes idle its resources are reclaimed. When activity returns, resources are resumed. Customers pay only for resources they used. However, scaling is currently merely reactive, not proactive, according to customers' workloads. Therefore, resources may not be immediately available when a customer comes back online after a prolonged idle period. In this work, we focus on reducing this delay in resource availability by predicting the pause/resume patterns and proactively resuming resources for each database. Furthermore, we avoid taking away resources for short idle periods to relieve the back-end from ineffective pause/resume workflows. Results of this study are currently being used worldwide to find the middle ground between quality of service and cost of operation.

PVLDB Reference Format:
Olga Poppe, Qun Guo, Willis Lang, Pankaj Arora, Morgan Oslake, Shize Xu, and Ajay Kalhan. Moneyball: Proactive Auto-Scaling in Microsoft Azure SQL Database Serverless. PVLDB, 15(6): 1279-1287, 2022.
doi:10.14778/3514061.3514073

1 INTRODUCTION

Microsoft Azure SQL Databases [5], Google Cloud SQL Databases [13], and Amazon RDS for SQL Server [3] are the leading relational database service providers in the cloud. They deploy automatic, fully managed databases to guarantee high Quality of Service (QoS) to their customers, while controlling Cost of Goods Sold (COGS).

Azure SQL Database serverless automatically scales resources based on demand and bills for the amount of resources used per second [7]. However, resumes and pauses are currently merely reactive, meaning that they do not take typical resource usage patterns into account. Therefore, serverless compute can introduce delays in resource availability after idle periods. Consequently, serverless compute may be less suitable for time-critical applications than provisioned compute that allocates a fixed amount of resources [6].

In this work, we aim to overcome the reactive nature of serverless compute by proactively resuming resources based on historical resume patterns. Furthermore, if pauses are short, the availability time of resources is too fragmented for effective reuse. Thus, we aim to relieve the back-end from short pauses.

Proceedings of the VLDB Endowment, Vol. 15, No. 6 ISSN 2150-8097.
doi:10.14778/3514061.3514073

Challenges. Optimizing Azure SQL Database serverless tier is a challenging endeavour for the following reasons.

(1) *Large search space of tunable parameters.* Proactive auto-scale of resources depends on many tunable parameters. The search space is prohibitively expensive to be explored exhaustively. Therefore, we identify trends of how these parameters influence the results and choose a reasonable set of parameters. The choice of parameters usually involves a trade-off between QoS and COGS [46]. For example, resuming resources in advance will guarantee high QoS, but waste COGS until these resources are used. We will discuss the trade-offs, while exploring the search space of parameters.

(2) *Opposing optimization objectives.* We aim to enable proactive resumes and avoid short pauses. However, these are opposing goals. Indeed, increasing the number of proactive resumes will also increase the number of wrong resumes, i.e., the customer did not come online as expected. Wrong resumes will in turn increase the number of pauses, some of which will be short. Also, reducing the number of short pauses will reduce the number of resumes, making the task of correct proactive resume harder because of fewer historical resumes. Solving this catch-22 is the goal of this work.

(3) *Changed resource usage patterns.* Resource usage patterns on serverless compute changed compared to provisioned compute. For example, provisioned databases are typically short-lived and often underutilized [33, 36, 39, 46]. In contrast to that, half of serverless databases existed over three weeks and half of idle periods are within a few hours (Figures 6 and 10(a)). At the same time, we observe certain similarities. For example, only a negligible percentage of provisioned or serverless databases follow a strict daily or weekly pattern. Therefore, we need to determine which lessons learned on provisioned compute can be transferred to serverless compute.

State-of-the-Art Techniques. While there are approaches to demand-driven auto-scale of resources in the cloud [27–30, 32, 37, 43–45], to the best of our knowledge, none of them addresses all challenges described above. In particular, they do not focus on achieving the contradictory goals of enabling proactive resume to guarantee high QoS, while reducing the number of short pauses to keep the operational costs low. Some of the existing approaches studied resource allocation on provisioned compute [26, 33, 36, 39, 40, 46] and we will transfer lessons learned from provisioned compute to serverless compute, when possible.

Our Proposed Solution. To address the challenges above, we propose the Moneyball approach that finds the middle ground between the contradictory goals of enabling proactive resume, while reducing the number of short pauses.[1]

[1] Similarly to the book and the movie "Moneyball" [1, 34], we apply statistical methods to achieve good results, while minimizing costs. However, we do not borrow statistical methods from this book.

To guarantee high QoS, we reduce delays in resource availability on login. To this end, we predict the pause/resume patterns and make recommendations when to proactively resume resources for each database. We compare probabilistic and predictive approaches to proactive resume and tune the key parameters to increase the number of correct proactive resumes, while reducing the operational costs due to wrong proactive resumes and the wait time intervals until the proactively resumed resources are used.

To reduce the back-end workload, we avoid short pauses. We compare two alternative solutions. One, we restrict the number of pauses per database and day (called a budget-based solution). Two, we introduce a wait time interval (called logical pause) before we scale resources down (called physical pause). We consider greedy and predictive approaches and compare their results to the optimal result. We tune the main parameters to reduce the number of short pauses and the costs due to idle resources during avoided pauses.

Contributions. Cloud service providers have recently evolved from provisioned to serverless compute [2, 7–9, 11, 14, 15, 17, 22, 23]. All of them face the challenge of provisioning resources only when needed. We believe that Moneyball generalizes to the cloud model in any company. Its key contributions are the following.

(1) We define the two-dimensional Moneyball problem space. We propose a visual way to compare our proposed solutions to the optimum and their impact on QoS and COGS.

(2) We summarize the main lessons learned during a decade of analysis of provisioned SQL databases. We transfer this learning to serverless compute, while solving the Moneyball problem. In particular, we select features and compare ML models to heuristics with respect to accuracy and maintenance overhead.

(3) We analyze production telemetry of serverless SQL databases with respect to their lifespan and typical resource usage patterns during half a year in tens of Azure regions where tens of thousands of serverless databases are currently deployed. Given the size and scope of this analysis, we believe that the usage patterns we observed represent the behaviors of any serverless databases.

(4) We enable proactive resume based on historical resume patterns per database. Up to 80% of resumes are proactive and correct within several hours for long-lived databases that existed at least 3 weeks. 99% of long-lived databases benefit from proactive resumes.

(5) We avoid short pauses by logically pausing a database that becomes idle before scaling its resources down. Logical pause is a simple, effective, and flexible technique that avoids up to half of pauses. 49% of databases benefit from this workload reduction.

2 MONEYBALL PROBLEM

Provisioned vs Serverless Compute. Resources of Azure SQL Databases are currently allocated in two ways.

Provisioned compute allocates a fixed amount of resources that does not change over time unless the customer explicitly requests a different amount [6]. Resources of a database s are resumed during the entire life time of s (Figure 1(a) and Table 1). However, rigorous telemetry analysis reveals that these resources are often underutilized [25, 26, 33, 36, 40, 46]. There are extensive idle periods during which resources are wasted unless customers manually scale resources down. This manual resource scaling is labor-intensive, time-consuming, error-prone, neither scalable, nor durable.

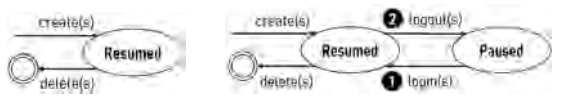
Table 1: Table of notations

Notation	Description
S	Set of databases, $s \in S$
d	Weekday (e.g., Wednesday)
W	Set of time windows within a day, $w \in W$
θ	Threshold
k	Budget
l	Duration of logical pause in hours
$H(s)$	Historical data of s
$h(s,d)$	Number of d's in $H(s)$
$r(s,d,w)$	Number of d's on which s was resumed during w in $H(s)$
$p(s,d,w)$	Probability of resume of s on d during w
$H(s,d,w)$	Historical data of s on d during w
$P(s,d)$	Predicted pause/resume pattern of s on d
$Predict(s,d)$	Time complexity of predicting the pause/resume pattern of s on d
$cost$	COGS per vCore per hour in dollars
$vcores(s)$	Maximum vCores of s
$pauses(s)$	Total duration of all pauses of s in hours without proactive resume
$wait(s)$	Total wait time in hours until proactively resumed resources of s are used
$avoided(s)$	Total duration of avoided pauses of s in hours
$allowed(s)$	Number of pauses of s that are longer than l
$idle(s,l)$	s is idle during l
$create(s)$	s is created
$delete(s)$	s is deleted
$login(s)$	Customer logs in to s
$logout(s)$	Customer logs out of s
$login(s).time$	Time stamp of $login(s)$
$p.start$	Start time stamp of a pause p

To overcome these limitations, *serverless compute* was recently introduced [7]. The resources of a database s are automatically scaled based on demand. If the serverless database is online, then scaling generally occurs with low latency since some resources remain allocated which helps mitigate the performance impact from compute warmup. However, if the serverless database is paused, then more or even all compute resources may be deallocated which elongates the latency to subsequently resume the database when workload activity returns. This paper focuses on minimizing the latency to resume a database since that has the greater performance impact between these two auto-scaling scenarios.

When the customer logs in, resources are resumed (❶ in Figure 1(b)). When the customer logs out, resources are paused for this database, reclaimed, and possibly assigned to other active databases (❷). Customers are billed per second only when resources are resumed for their databases. Thus, serverless compute minimizes both the waste of resources and the costs for the customers. However, serverless compute can be further optimized as described below.

Reactive vs Proactive Resume. Transitions between paused and resumed states are not instantaneous (Figure 1(b)). A resume workflow assigns resources to a database that becomes active, while



(a) Provisioned compute (b) Reactive auto-scale on serverless compute

Figure 1: Life cycle of a database

a pause workflow takes resources away from a database that becomes idle (Figure 2). Currently, resumes are merely reactive, not proactive. Delays in resource availability may occur after long idle periods. These delays make serverless compute less suitable for time-critical applications than provisioned compute [6].

In this work, we aim to reduce these delays by proactively resuming resources based on historical resume patterns per database. For example, if a database is usually resumed during a window w, we can proactively resume resources at the beginning of w.

Definition 2.1. (**Correct Proactive Resume**) A proactive resume of a database s within a window w is *correct* if the resources of s are used within w. Otherwise, a proactive resume is *wrong*.

However, if we proactively resume too far in advance, resources will stay idle until the customer logs in and uses them. COGS are wasted during these idle intervals. COGS are also wasted due to wrong proactive resumes. We aim to enable proactive resume, while keeping its operational cost low (Definition 4.6).

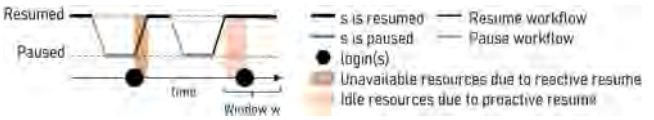


Figure 2: Reactive vs proactive resume

Effective vs Ineffective Pause. A pause is ineffective for short idle periods because the availability time of resources is too fragmented for effective reuse (Figure 3). We aim to relieve the back-end from frequent pause/resume workflows.

Definition 2.2. (**Ineffective Pause**) Given a threshold l, a pause is called *ineffective* if its duration is within l.

However, if we do not pause for long idle intervals, resources and COGS will be wasted. We aim to avoid up to half of pauses, while reducing the operational costs (Definitions 5.2 and 5.4).

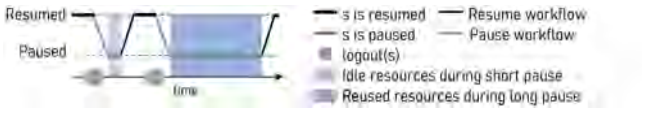


Figure 3: Effective vs ineffective pause

Moneyball Problem Statement. In this work, we aim to achieve the following three goals: (1) Maximize the number of correct proactive resumes, (2) Minimize the number of short pauses, and (3) Minimize the operational costs of these two optimization techniques.

3 TRANSFER LEARNING FROM PROVISIONED TO SERVERLESS COMPUTE

The resource usage patterns of provisioned Azure SQL Databases have been rigorously studied for over a decade [26, 33, 36, 39, 40, 46]. In this section, we briefly summarize the main lessons learned that can be helpful to solve the Moneyball problem and transfer this learning to serverless compute, when possible.

3.1 Features

Provisioned Compute. Historical load is an indicator of the future load per database. Some databases have stable load. Others follow a business pattern. At least three weeks of historical data are required to make a reliable load prediction [36, 40, 46]. Resource usage patterns may be different for databases with different editions (e.g., premium, standard), performance levels (i.e., number of Database Transaction Units (DTU) [6]), and Azure regions [33, 36, 39]. Furthermore, resource usage patterns may change over time.

Serverless Compute. To capture result variation between different regions and weeks, we analyzed half a year of production telemetry from tens of Azure regions where tens of thousands serverless databases are currently deployed. We included all features that can be useful for the prediction of pause/resume behavior in our analysis. They are: timestamp in seconds, database identifier, database state (1 means resumed, -1 means paused), duration of time intervals during which this database was resumed or paused, database compute capacity in maximum vCores, database creation and deletion timestamps, and Azure region.

3.2 ML Models

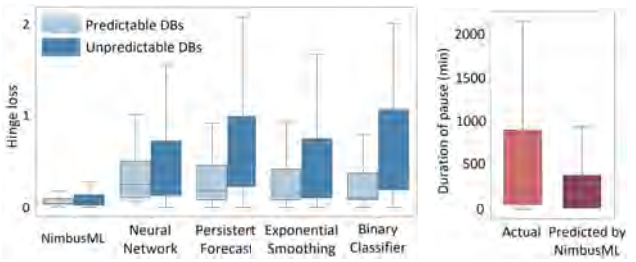
Provisioned Compute. In our prior research [40, 41], we predicted the load of provisioned databases using ARIMA [4], Prophet [21], NimbusML [18], Neural Network [12], Exponential Smoothing [10] and the Persistent Forecast heuristic that uses the load on a given day as the prediction of the load on next day per database. ARIMA and Prophet do not scale to tens of thousands of databases in large Azure regions. While NimbusML is the most accurate model, the gain in accuracy is not significant compared to Persistent Forecast because provisioned databases fall into one of the following extremes: (1) Most databases are *easily predictable* even by Persistent Forecast because their load is stable or follows a pattern (Definitions 3.1 and 3.2). (2) Remaining databases are *hard to predict* even by advanced ML models because their load tends to be random.

Serverless Compute. To verify that this conclusion holds for serverless databases, we classified them by their typical pause/resume patterns into the following groups.

Definition 3.1. (**Stable Database**) Given historical data $H(s)$ of a database s and a threshold θ, s is called *stable* if s is either resumed or paused at least θ% of the time in $H(s)$. Otherwise, s is *unstable*.

Definition 3.2. (**Pattern**) Let s be an unstable database, $H(s)$ be the historical data of s, d be a weekday, w be a window, and θ be a threshold. s follows a *pattern* if at least θ% of its resumes and pauses happen within the window w on each weekday d in $H(s)$.

Definition 3.3. (**Predictable Database**) A database s is called *predictable* if s is stable or follows a pattern. Otherwise, s is called *unpredictable*.



(a) Accuracy of ML models (b) Pause duration

Figure 4: Results of ML models

We randomly sampled several thousands of serverless databases in one Azure region. Similarly to provisioned databases, most serverless databases are predictable even under strict constraints. Indeed, 74% of serverless databases are stable at least 90% of the time. 3% follow a pattern within 15 minutes at least 90% of the time. Remaining 23% of databases are unpredictable.

We trained ML models on three weeks of historical data and predicted the load on the following day per database. Given that our input data is binary (Section 3.1), we measure hinge loss of NimbusML [18], Neural Network [12], Exponential Smoothing [10], ML.NET Binary Classifier [16], and Persistent Forecast for each class of databases in Figure 4(a). As expected, all models are more accurate for predictable databases than for unpredictable databases. Similarly to provisioned compute, NimbusML is the most accurate among these models for both classes of databases. Thus, we include NimbusML in our detailed analysis below.

4 PROACTIVE RESUME

In this section, we analyze resume patterns per database over time, make recommendations when to proactively resume a database, and evaluate the effectiveness of these recommendations.

4.1 Proactive Resume Algorithms

Example 4.1. The database in Figure 5 is unpredictable by Definition 3.3. However, a closer look reveals that this database is usually resumed between 5:40AM and 9:20AM on Wednesdays. These resumes are highlighted by red arrows. Only one expected resume is missing on 2/17. Next, we describe how to detect such recurring resumes and make them proactive.

Definition 4.2. (**Probability of Resume**) Let $H(s)$ be the historical data of a database s, $h(s,d)$ be the number of weekdays d in $H(s)$, and $r(s,d,w)$ be the number of d's on which s was resumed during a window w in $H(s)$ (Table 1). The *probability of resume* of s on d during w is computed as $p(s,d,w) = \frac{r(s,d,w)}{h(s,d)}$ [20].

Example 4.3. Given eight weeks of history in Figure 5, the probability of resume of this database s on Wednesday between 5:40 and 9:20 is $p(s, Wednesday, [5{:}40, 9{:}20]) = \frac{7}{8} = 0.875$.

Definition 4.4. (**Probabilistic Resume Recommendation**) Given a threshold θ, we recommend to *proactively resume* a database s on a weekday d at the beginning of a window w if $p(s,d,w) \geq \theta$.

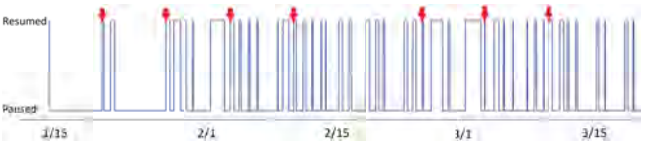


Figure 5: Recurring resumes

Probabilistic Resume computes resume recommendations R on a weekday d based on historical data of databases S. For each database $s \in S$ and window $w \in W$, Algorithm 1 adds a recommendation $[s,d,w]$ to proactively resume a database s on a weekday d at the beginning of a window w to the set of results R if the probability of resume $p(s,d,w)$ satisfies the threshold θ.

Complexity. Let $|S|$ be the number of databases in S, $|W|$ be the number of windows in W, and $|H(s,d,w)|$ be the number of tuples in historical data per database, weekday, and window. The time complexity of Algorithm 1 is $O(|S| \times |W| \times |H(s,d,w)|)$. Its space complexity is determined by the number of databases $|S|$, the number of tuples in historical data per database $|H(s)|$, and the number of recommendations per database in R. The number of results per database is in tern determined by the number of windows $|W|$. In summary, the space complexity is $O(|S| \times (|H(s)| + |W|))$.

Algorithm 1 Probabilistic proactive resume

Input: Historical data of databases S, set of windows W within one day, probability threshold θ
Output: Set of resume recommendations R on a weekday d

```
for each s ∈ S do
    for each w ∈ W do
        if p(s,d,w) ≥ θ then R ← R ∪ [s,d,w]
return R
```

Definition 4.5. (**Predictive Resume Recommendation**) Given the predicted pause/resume pattern $P(s,d,w)$ for a database s on a weekday d during a window w, we recommend to *proactively resume* s on d at the beginning of w if $\exists resume \in P(s,d,w)$.

Predictive Resume algorithm is analogous to Algorithm 1 except that it consumes predicted pause/resume patterns and detects predicted resumes per Definition 4.5. While any ML model can be plugged into this algorithm to predict pause/resume patterns, we chose NimbusML since it is the most accurate model in Figure 4(a).

Complexity. Predictive resume introduces the overhead of predicting the pause/resume pattern per database and day, denoted $Predict(s,d)$. Thus, the time complexity is $O(|S| \times (Predict(s,d) + |W| \times |P(s,d,w)|))$. Predictive resume also stores the predicted pause/resume pattern per database and day, denoted $P(s,d)$. The space complexity is $O(|S| \times (|H(s)| + |P(s,d)| + |W|))$.

4.2 Middle Ground between QoS and COGS

While proactive resumes improve QoS, they also shorten pauses during which resources can be reused and COGS can be saved. Also, some proactive resumes will be wrong unavoidably. COGS are wasted due to wrong resumes as well. Definition 4.6 quantifies the operational cost of proactive resume.

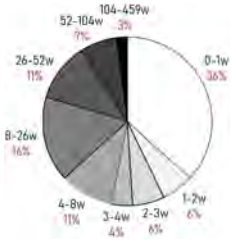
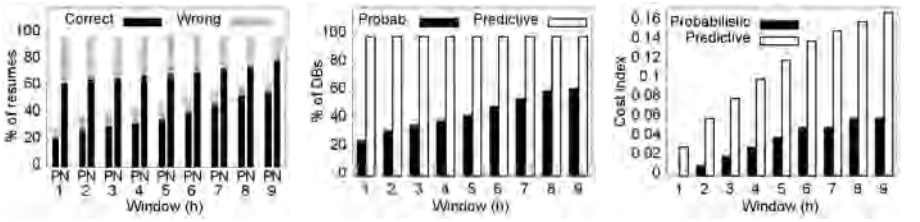


Figure 6: Lifespan



(a) Proactive resumes: Probabilistic resume (P) vs predictive resume (N)

(b) Benefited databases

(c) Resume cost index

Figure 7: Varying size of time window

Definition 4.6. (**Resume Cost Index**) Let $pauses(s)$ be the total duration of all pauses of a database s in hours without proactive resume. Let $vcores(s)$ be the maximum vCores of s and $cost$ be COGS per vCore per hour in dollars. The total cost savings are:

$$Total\ cost\ savings = \sum_{s \in S} pauses(s) \times vcores(s) \times cost \quad (1)$$

Let $wait(s)$ be the total wait time in hours until proactively resumed resources of a database s are used. The wasted cost is:

$$Wasted\ cost = \sum_{s \in S} wait(s) \times vcores(s) \times cost \quad (2)$$

Resume cost index corresponds to the ratio of the wasted cost to the total cost savings.

The cost index depends on several tunable parameters such as the size of the window and the length of historical data. We now experimentally find the middle ground between QoS and COGS, while enabling proactive resume.

In Figure 6, we measure the percentage of databases per their lifetime in weeks. Half of databases existed at least 3 weeks and thus have enough history to make a reliable prediction (Section 3).

Definition 4.7. (**Long-Lived Database**) A database is *long-lived* if it exists at least three weeks. Otherwise, it is *short-lived*.

Setup. In Figures 7–10 and 12, results are shown for several thousands of randomly sampled serverless long-lived databases in one Azure region. Unless stated otherwise, the length of historical (training) data is 3 weeks. The length of validation time interval is 1 day. Default size of the window is 5 hours. The window slides every 10 minutes. Default probability threshold is 0.9.

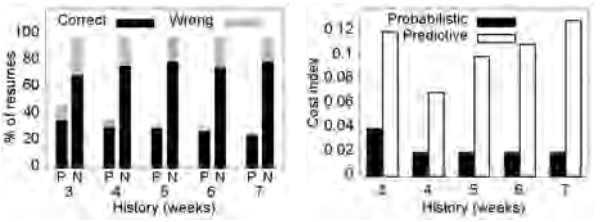
Size of the Window. In Figure 7, we vary the size of the window from 1 to 9 hours and measure the percentage of correct and wrong proactive resumes among all resumes (Definition 2.1), the percentage of database that have correct proactive resumes, and the resume cost index (Definition 4.6)

The percentages of correct resumes increase from 22 to 56 for probabilistic resume as the window grows in Figure 7(a). These percentages increase from 63 to 80 for predictive resume. Probabilistic resume benefits 25 to 62% of databases as the window grows in Figure 7(b). Independently from the size of the window, predictive resume benefits 99% of databases. The percentages of correct resumes and benefited databases is up to 3X higher for predictive resume than for probabilistic resume.

Unfortunately, the cost index also grows with the window since proactively resumed resources stay idle longer. Probabilistic resume has up to 5X fewer wrong resumes. Therefore, its cost index is up to 5X lower than the cost index of predictive resume in Figure 7(c).

Length of Historical Data. In Figure 8, we vary the length of historical data from 3 to 7 weeks. Given 3 weeks of history, 36% of resumes are proactive and correct, 43% of databases have correct proactive resumes, 12% of resumes are proactive and wrong, and the cost index is 4% for probabilistic resume. These percentages decrease as the length of historical data grows.

There is are no clear trends for the results of predictive resume across weeks. 70 to 80% of resumes are proactive and correct. 99% of databases have correct resumes. 18 to 29% of resumes are wrong. The cost index ranges from 7 to 12%. Similarly to Figure 7, predictive resume has up to 3X more correct resumes and benefited databases than probabilistic resume. Predictive resume also has up to 18X more wrong resumes. Thus, its cost index is up to 10X higher than the cost index of probabilistic resume.



(a) Proactive resumes: Probabilistic resume (P) vs predictive resume (N)

(b) Resume cost index

Figure 8: Varying length of historical data

Summary. Most resumes are proactive and correct within a few hours for long-lived databases. Most long-lived databases benefit from this QoS optimization. Cost index is low for short windows.

5 AVOIDING INEFFECTIVE PAUSES

Pauses are ineffective for short idle periods. Indeed, no COGS are saved and unnecessary pause/resume workloads are introduced. To alleviate these workloads from the back-end, we avoid ineffective pauses by restricting the number of pauses (called *budget*) and delaying pauses (called *logical pause*).

5.1 Budgeting Algorithms

One straightforward idea that comes to mind is to restrict the number of pauses per database and day and prioritize long pauses.

Definition 5.1. (**Budget**) *Budget* k is the number of allowed pauses per database s and window w.

A given budget can be spent in different ways as defined below.

Greedy Budget allows the first k pauses and avoids all following pauses per database and day. Let $logout_i(s).time$ and $login_{i+1}(s).time$ be the time stamps of two consecutive logout and login events for a database s. For each database s, Algorithm 2 stores the beginning $logout_i(s).time$ and the end $login_{i+1}(s).time$ of each avoided pause in the set of results R.

Complexity. Given the number of logouts $|logout(s,d)|$ for a database s on a weekday d, the time complexity of Algorithm 2 is $O(|S| \times |logout(s,d)|)$. Since avoided pauses are stored in the set of results R, the space complexity is $O(|S| \times (|logout(s,d)| - k))$.

Algorithm 2 Greedy budget

Input: Login/logout events of databases S on weekday d, budget k
Output: Set of avoided pauses R on a weekday d

```
1: for each s ∈ S do n ← 1
2:     for each logout_i(s) do
3:         if n > k then
4:             R ← R ∪ [s, logout_i(s).time, login_{i+1}(s).time]
5:         else n ← n + 1
6: return R
```

Algorithm 2 disregards the duration of avoided pauses. In the worst case, it spends the budget on early short pauses and avoids later long pauses which makes greedy budget expensive.

Definition 5.2. (**Pause Cost Index**) Let $avoided(s)$ be the duration of avoided pauses in hours for a database s. Other notations are summarized in Table 1. Then, the wasted cost is computed as:

$$Wasted\ cost = \sum_{s \in S} avoided(s) \times vcores(s) \times cost \quad (3)$$

The *pause cost index* is defined as the ratio of the wasted cost (Equation 3) to the total cost savings (Equation 1).

Predictive Budget prioritizes predicted long pauses over predicted short pauses, while spending the budget. For each database $s \in S$, Algorithm 3 consumes the predicted pause/resume pattern $P(s,d)$, extracts the longest k predicted pauses, and avoids all actual pauses that do not start within a given time delta δ of a long predicted pause.

Complexity. Predictive budget introduces the overhead of predicting the pause/resume pattern per database and day $Predict(s,d)$, sorting the predicted pauses by duration in $O(|P(s,d)| \log |P(s,d)|)$ time, and comparing the beginnings of k longest predicted pauses to the timestamps of actual logouts in $O(|logout(s,d)| \times k)$ time. The time complexity is $O(|S| \times (Predict(s,d) + |P(s,d)| \log |P(s,d)| + |logout(s,d)| \times k))$. Algorithm 3 stores the historical data, the predicted pause/resume pattern, and the avoided pauses per database. Its time complexity is $O(|S| \times (|H(s)| + |P(s,d)| + |logout(s,d)| - k))$.

Algorithm 3 Predictive budget

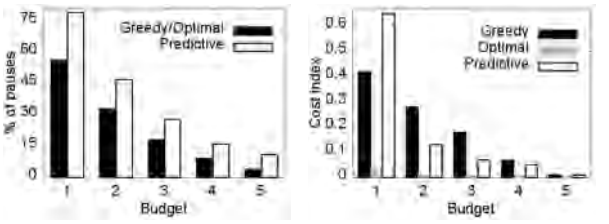
Input: Login/logout events of databases S on weekday d, predicted pause/resume patterns of S on d, budget k, window $w = 2 \times \delta$
Output: Set of avoided pauses R on a weekday d

```
1: for each s ∈ S do pauses ← getLongPauses(P(s,d), k), n ← 1
2:     for each logout_i(s) do
3:         if n > k or ∄p ∈ pauses such that
4:         logout_i(s).time − δ ≤ p.start ≤ logout_i(s).time + δ then
5:             R ← R ∪ [s, logout_i(s).time, login_{i+1}(s).time]
6:         else n ← n + 1
7: return R
```

Optimal Budget. To evaluate the effectiveness of the greedy and predictive budgets, we compare them to the optimal budget that avoids the top k shortest pauses per database.

Value of Budget. The percentages of avoided pauses and the cost index depend on the value of k which we vary in Figure 9. Greedy and optimal budget avoid 56 to 4% of pauses as budget grows from 1 to 5 in Figure 9(a). 56 to 2% of databases have avoided pauses for budget 1 to 5. While greedy budget disregards the duration of avoided pauses, optimal budget avoids the shortest k pauses per database and day. Thus, the cost of optimal budget is one order of magnitude lower than the cost of greedy budget in Figure 9(b).

Predictive budget delays pausing a database until long predicted pauses to reduce the time intervals during which resources are idle and COGS are wasted. However, if the start or duration of the longest k pauses per database and day are predicted wrong, then the predictive algorithm does not spend the available budget and avoids up to 5X more pauses than the greedy algorithm in Figure 9(a). Nevertheless, the cost of the predictive algorithm is up to 3X lower than the cost of the greedy algorithm for budget 2 to 4 because the predictive algorithm prioritizes long pauses while spending the budget (Figure 9(b)). Unfortunately, predictive budget does not guarantee lower cost compared to greedy budget. In fact, the cost of the predictive algorithm is 55% higher than the cost of the greedy algorithm for budget 1 in Figure 9(b).



(a) Avoided pauses (b) Pause cost index

Figure 9: Budget

Budget can be defined at different system granularities (e.g., per database, per tenant ring, per cluster) and for different windows (e.g., daily, weekly, monthly). However, we observed similar results to Figure 9. We do not consider global budget because resources are not shared across clusters in Azure.

Summary. Greedy budget disregards the duration of avoided pauses. Thus, its cost index is one order of magnitude higher than

the cost index of optimal budget. Predictive budget does not always spend the available budget and thus does not guarantee lower cost compared to greedy budget.

5.2 Logical Pause-Based Algorithms

Another simple idea is to wait for the customer to come back online before taking resources away from her database.

Definition 5.3. (**Logical Pause, Physical Pause**) Let $logout_i(s).time$ and $login_{i+1}(s).time$ be the time stamps of two consecutive logout and login events for s. Let t be the time stamp when resources are taken away from s such that $logout_i(s).time < t < login_{i+1}(s).time$. The time interval $(logout_i(s).time, t)$ is called *logical pause*. The time interval $[t, login_{i+1}(s).time)$ is called *physical pause*.

Greedy Logical Pause logically pauses a database s for the time interval l when the customer logs out.

Complexity. The time complexity of Algorithm 4 is the same as for Algorithm 2. Its space complexity is $O(|S| \times |logout(s,d)|)$.

Algorithm 4 Greedy logical pause

Input: Login/logout events of databases S on weekday d, duration of logical pause l
Output: Set of avoided pauses R on a weekday d

```
1: for each s ∈ S do
2:     for each logout_i(s) do
3:         if logout_i(s).time + l ≤ login_{i+1}(s).time then
4:             R ← R ∪ [s, logout_i(s).time, logout_i(s).time + l]
5:         else R ← R ∪ [s, logout_i(s).time, login_{i+1}(s).time]
6: return R
```

During logical pause, resources are still available in case the customer comes back online. In this way, we reduce delays in resource availability, while avoiding short pauses. However, resources are idle during avoided pauses (Line 5). In addition, pauses that are longer than l are shortened by greedy logical pauses (Line 4). This makes greedy pause expensive.

Definition 5.4. (**Greedy Pause Cost Index**) Let l be the duration of logical pause in hours, $avoided(s)$ be the duration of avoided pauses in hours for a database s, and $allowed(s)$ be the number of pauses of s that are longer than l. Other notations are summarized in Table 1. Then, the wasted cost is computed as follows:

$$Wasted\ cost = \sum_{s \in S} (avoided(s) + l \times allowed(s)) \times vcores(s) \times cost$$

The *greedy pause cost index* is defined as the ratio of the wasted cost to the total cost savings (Equation 1).

Predictive Logical Pause avoids predicted short pauses without reducing the duration of predicted long pauses. For each database $s \in S$, Algorithm 5 consumes the predicted pause/resume pattern $P(s,d)$ and computes the maximal duration of predicted pauses within the time delta δ of each logout. If this maximal duration is shorter than logical pause l, then this pause is avoided.

Complexity. Algorithm 5 predicts the pause/resume pattern per database and day and computes the maximal duration of predicted pauses that start within the time delta δ of each logout. Its time complexity is $O(|S| \times (Predict(s,d) + |logout(s,d)| \times |P(s,d)|))$. Algorithm 5 stores the historical data, the predicted pause/resume pattern, and the avoided pauses per database. Its space complexity is $O(|S| \times (|H(s)| + |P(s,d)| + |logout(s,d)|))$.

Algorithm 5 Predictive logical pause

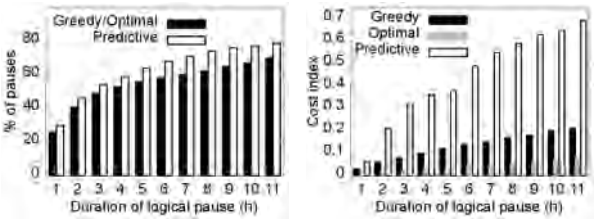
Input: Login/logout events of databases S on weekday d, predicted pause/resume patterns of S on d, duration of logical pause l, window $w = 2 \times \delta$
Output: Set of avoided pauses R on a weekday d

```
1: for each s ∈ S do
2:     for each logout_i(s) do
3:         w_i ← [logout_i(s).time − δ, logout_i(s).time + δ]
4:         maxDuraiton ← getMaxPauseDuration(P(s,d), w_i)
5:         if maxDuration < l then
6:             R ← R ∪ [s, logout_i(s).time, login_{i+1}(s).time]
7: return R
```

Optimal Logical Pause. To evaluate the effectiveness of the greedy and predictive logical pause, we compare them to the optimal logical pause that avoids all pauses that are shorter than l.

Duration of Logical Pause. The number of avoided pauses and the cost index depend on the duration of logical pause l that we vary in Figure 10. The greedy and optimal algorithms avoid 26 to 70% of pauses in Figure 10(a) and benefit 33 to 58% of databases as the duration of logical pause increases from 1 to 11 hours. The cost index of the greedy algorithm is up to 6X higher than the cost index of the optimal algorithm in Figure 10(b).

Since predicted pauses tend to be shorter than the actual pauses (Figure 4(b)), the predictive algorithm avoids up to 19% more pauses than the greedy algorithm in Figure 10(a). Thus, the cost index of the predictive algorithm up to 4X higher than the cost index of the greedy algorithm in Figure 10(b).



(a) Avoided pauses (b) Pause cost index

Figure 10: Logical pause

Summary. Greedy logical pause is a simple, flexible, and effective technique to avoid short pauses. Most databases benefit from this optimization technique at relatively low cost.

6 PUTTING IT ALL TOGETHER

In this section, we summarize how proactive resume and logical pause work together to proactively scale serverless databases. We also evaluate the impact of these optimization techniques.

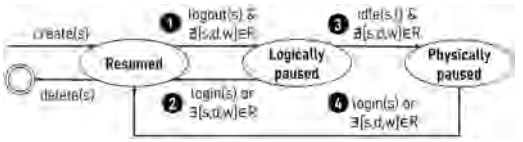


Figure 11: Proactive auto-scale on serverless compute

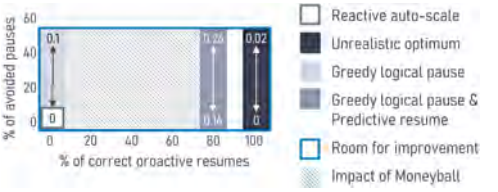


Figure 12: Moneyball problem space

Proactive Auto-Scale on Serverless Compute. Figure 11 formalizes the life cycle of a serverless proactively scaled database (compare to Figure 1). Let d be the current weekday and w be the current window. Other notations are summarized in Table 1.

The database s stays resumed as long as s is active or there is a recommendation to keep s proactively resumed on d during w, denoted $\exists[s, d, w] \in R$. If there is no such recommendation, then s is logically paused once the customer logs out (❶ in Figure 11). s stays logically paused for at most the time interval l. During logical pause, if the customer logs in or there is a recommendation to proactively resume s on d during w, then s is resumed (❷). If s stays idle during logical pause l and no resume is expected during on d during w, then s is physically paused (❸). s stays physically paused until s is resumed once the customer logs in or there is a recommendation to proactively resume s on d during w (❹).

Impact of Moneyball. Figure 12 illustrates the two-dimensional problem space where each dimension corresponds to the optimization technique enabled by Moneyball. X-axis represents the percentage of correct proactive resumes, while Y-axis depicts the percentage of avoided pauses. Rectangles represent alternative solutions and numbers correspond to their respective cost indexes.

In reactive approach, no resumes are proactive, no pauses are avoided, and thus no COGS are wasted (i.e., the cost index is 0). This case is shown as a white rectangle in Figure 12.

Ideally, all resumes are proactive and correct. In addition, up to half of pauses are avoided and these avoided pauses are the shortest to reduce resource idleness and wasted COGS. The cost index of the optimal solution is up to 0.02 (Definition 5.2). The range of these unrealistic optimal solutions is shown as a black rectangle. The area between the reactive approach and the optimal solution, highlighted by blue frame, is the potential room for improvement.

To avoid ineffective pauses, we introduce a wait time interval, called logical pause, before scaling resources down. Given that resources are idle during logical pauses, this solution wastes COGS. The number of avoided pauses and the cost index depend on the duration of logical pauses (Figure 10). For example, if logical pause is 4 hours, 53% of pauses are avoided and the cost index is 0.1 (Definition 5.4). The spectrum of logical-pause-based solutions is shown as a light gray rectangle.

Up to 80% of all resumes are proactive and correct within several hours for long-lived databases. Due to wrong resumes and wait time until the proactively resumed resources are used, the cost index is 0.16 (Definition 4.6). Combining proactive resume with logical pause makes up to 80% of resumes proactive and correct for long-lived databases, while still avoiding up to half of pauses. This combined Moneyball approach is shown as a dark gray rectangle. Its cost index is 0.26 which we consider to be reasonable cost for these optimization techniques. The striped area between the reactive approach and Moneyball represents the impact of this work.

7 RELATED WORK

Self-driving databases [19, 38] in general and demand-driven auto-scale of resources [26–30, 32, 37, 43–45] in particular have become popular research directions in the recent years. However, some of these state-of-the-art approaches are merely reactive [26–28]. In contrast, our Moneyball approach is proactive based on typical resource usage patterns per database (Section 4).

Some approaches avoid and mitigate under-estimation errors [44], reconfigure databases based on predicted load [45], or benchmark the efficiency of a cloud service [36]. These mechanisms are orthogonal to the Moneyball problem (Section 2).

Other approaches focus on load analysis [25, 32, 35, 42], load prediction using machine learning and other techniques [24, 29–31, 33, 36, 39, 40, 43, 46], or learning a relationship between available resources and performance [37]. We transferred learning from these approaches to Azure SQL Database serverless to solve the Moneyball problem (Sections 3–5).

While several state-of-the-art approaches focus on solving the trade-off between QoS and COGS in the cloud [27, 29, 33, 36, 39, 43, 44, 46], none of them achieves the contradictory goals of enabling proactive resume to guarantee high QoS and avoiding short pauses to alleviate this workload, while controlling operational costs at the same time. This is the key contribution of our Moneyball approach.

8 CONCLUSIONS

The Moneyball approach introduces two optimization techniques of Microsoft Azure SQL Database serverless. (1) To reduce delays in resource availability, we predict resume patterns per database over time and proactively resume resources. (2) To reduce the back-end workload, we avoid short pauses by logically pausing a database that becomes idle before scaling its resources down. We compared several algorithms and tuned their key parameters to keep the operational cost of these optimization techniques low. Results of this study are used in production in all Azure regions.

ACKNOWLEDGMENTS

The authors thank Ehi Nosakhare and Karthik Rajendran for their hard work predicting the load of provisioned SQL databases. Their findings guided our approach on serverless compute. We also thank Carlo Curino, Yiwen Zhu, and VLDB reviewers for their feedback.

REFERENCES

[1] 2011. *Moneyball (film).* Retrieved December 15, 2021 from https://en.wikipedia.org/wiki/Moneyball_(film)

[2] 2021. *Alibaba Cloud Function Compute.* Retrieved December 15, 2021 from https://www.alibabacloud.com/product/function-compute

[3] 2021. *Amazon RDS for SQL Server.* Retrieved December 15, 2021 from https://aws.amazon.com/rds/sqlserver

[4] 2021. *ARIMA.* Retrieved December 15, 2021 from https://pypi.org/project/pmdarima

[5] 2021. *Azure SQL Database.* Retrieved December 15, 2021 from https://azure.microsoft.com/en-us/products/azure-sql/database

[6] 2021. *Azure SQL Database pricing.* Retrieved December 15, 2021 from https://azure.microsoft.com/en-us/pricing/details/azure-sql-database

[7] 2021. *Azure SQL Database serverless.* Retrieved December 15, 2021 from https://docs.microsoft.com/en-us/azure/azure-sql/database/serverless-tier-overview

[8] 2021. *CockroachDB Serverless.* Retrieved December 15, 2021 from https://www.cockroachlabs.com/lp/serverless/

[9] 2021. *Databricks Serverless SQL.* Retrieved December 15, 2021 from https://databricks.com/blog/2021/08/30/announcing-databricks-serverless-sql.html

[10] 2021. *Exponential Smoothing.* Retrieved December 15, 2021 from https://www.statsmodels.org/stable/generated/statsmodels.tsa.holtwinters.ExponentialSmoothing.html

[11] 2021. *Fauna Serverless.* Retrieved December 15, 2021 from https://fauna.com/serverless

[12] 2021. *GluonTS.* Retrieved December 15, 2021 from https://gluon-ts.mxnet.io/

[13] 2021. *Google Cloud SQL.* Retrieved December 15, 2021 from https://cloud.google.com/sql

[14] 2021. *Google Serverless Computing.* Retrieved December 15, 2021 from https://cloud.google.com/serverless

[15] 2021. *IBM Cloud Functions.* Retrieved December 15, 2021 from https://www.ibm.com/cloud/functions

[16] 2021. *ML.NET Binary Trainer.* Retrieved December 15, 2021 from https://docs.microsoft.com/en-us/dotnet/api/microsoft.ml.trainers.fasttree.fastforestbinarytrainer

[17] 2021. *MongoDB Serverless.* Retrieved December 15, 2021 from https://www.mongodb.com/cloud/atlas/serverless

[18] 2021. *NimbusML.* Retrieved December 15, 2021 from https://docs.microsoft.com/en-us/python/api/nimbusml/nimbusml.timeseries.ssaforecaster

[19] 2021. *Oracle Autonomous Database.* Retrieved December 15, 2021 from https://www.oracle.com/autonomous-database/

[20] 2021. *Probability theory.* Retrieved December 15, 2021 from https://en.wikipedia.org/wiki/Event_(probability_theory)

[21] 2021. *Prophet.* Retrieved December 15, 2021 from https://facebook.github.io/prophet

[22] 2021. *Serverless on AWS.* Retrieved December 15, 2021 from https://aws.amazon.com/serverless/

[23] 2021. *Snowflake Serverless.* Retrieved December 15, 2021 from https://docs.snowflake.com/en/user-guide/admin-serverless-billing.html

[24] Rodrigo Calheiros, Enayat Masoumi, Rajiv Ranjan, and Rajkumar Buyya. 2014. Workload Prediction Using ARIMA Model and Its Impact on Cloud Applications' QoS. *IEEE Transactions on Cloud Computing* 3 (08 2014), 449–458.

[25] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *SOSP*. 153–167.

[26] Sudipto Das, Feng Li, Vivek R. Narasayya, and Arnd Christian König. 2016. Automated Demand-driven Resource Scaling in Relational Database-as-a-Service. In *SIGMOD*. 1923–1924.

[27] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: Resource-Efficient and QoS-Aware Cluster Management. *SIGPLAN Not.* 49, 4 (2014), 127–144.

[28] Avrilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. 2017. Dhalion: Self-Regulating Stream Processing in Heron. In *Proc. VLDB Endow.* 1825–1836.

[29] Zhenhuan Gong, Xiaohui Gu, and John Wilkes. 2010. PRESS: PRedictive Elastic ReSource Scaling for cloud systems. In *TNSM*. 9–16.

[30] Sadeka Islam, Jacky Keung, Kevin Lee, and Anna Liu. 2012. Empirical Prediction Models for Adaptive Resource Provisioning in the Cloud. *Future Generation Comp. Syst.* 28 (01 2012), 155–162.

[31] Arijit Khan, Xifeng Yan, Shu Tao, and Nikos Anerousis. 2012. Workload Characterization and Prediction in the Cloud: A Multiple Time Series Approach. In *IEEE Network Operations and Management Symposium*. 1287–1294.

[32] Cinar Kilcioglu, Justin M. Rao, Aadharsh Kannan, and R. Preston McAfee. 2017. Usage Patterns and the Economics of the Public Cloud. In *WWW*. 83–91.

[33] Willis Lang, Karthik Ramachandra, David J. DeWitt, Shize Xu, Qun Guo, Ajay Kalhan, and Peter Carlin. 2016. Not for the Timid: On the Impact of Aggressive over-Booking in the Cloud. *Proc. VLDB Endow.* 9, 13 (2016), 1245–1256.

[34] Michael Lewis. 2003. *Moneyball: The Art of Winning an Unfair Game.* W.W. Norton and Company.

[35] Asit K. Mishra, Joseph L. Hellerstein, Walfredo Cirne, and Chita R. Das. 2010. Towards Characterizing Cloud Backend Workloads: Insights from Google Compute Clusters. *SIGMETRICS Perform. Eval. Rev.* 37, 4 (March 2010), 34–41.

[36] Justin Moeller, Zi Ye, Katherine Lin, and Willis Lang. 2021. Toto - Benchmarking the Efficiency of a Cloud Service. In *SIGMOD*. 2543–2556.

[37] Pradeep Padala, Kai-Yuan Hou, Kang G. Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, and Arif Merchant. 2009. Automated Control of Multiple Virtualized Resources. In *EuroSys*. 13–26.

[38] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C. Mowry, Matthew Perron, Ian Quah, Siddharth Santurkar, Anthony Tomasic, Skye Toor, Dana Van Aken, Ziqi Wang, Yingjun Wu, Ran Xian, and Tieying Zhang. 2017. Self-Driving Database Management Systems. In *CIDR*.

[39] Jose Picado, Willis Lang, and Edward C. Thayer. 2018. Survivability of Cloud Databases - Factors and Prediction. In *SIGMOD*. 811–823.

[40] Olga Poppe, Tayo Amuneke, Dalitso Banda, Aritra De, Ari Green, Manon Knoertzer, Ehi Nosakhare, Karthik Rajendran, Deepak Shankargouda, Meina Wang, Alan Au, Carlo Curino, Qun Guo, Alekh Jindal, Ajay Kalhan, Morgan Oslake, Sonia Parchani, Vijay Ramani, Raj Sellappan, Saikat Sen, Sheetal Shrotri, Soundararajan Srinivasan, Ping Xia, Shize Xu, Alicia Yang, and Yiwen Zhu. 2020. Seagull: An Infrastructure for Load Prediction and Optimized Resource Allocation. *Proc. VLDB Endow.* 14, 2 (2020), 154–162.

[41] Olga Poppe, Alan Au, Aritra De, Raj Sellappan, Saikat Sen, Deepak Shankargouda, Meina Wang, Tayo Amuneke, Dalitso Banda, Ari Green, Manon Knoertzer, Ehi Nosakhare, Karthik Rajendran, Vijay Ramani, Soundararajan Srinivasan, Carlo Curino, Alekh Jindal, Yiwen Zhu, Qun Guo, Ajay Kalhan, Morgan Oslake, Shize Xu, Sonia Parchani, Sheetal Shrotri, and Ping Xia. 2020. Seagull: An Infrastructure for Load Prediction and Optimized Resource Allocation. Extended version.

[42] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. 2012. Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis. In *SOCC*. 1–13.

[43] Nilabja Roy, Abhishek Dubey, and Aniruddha Gokhale. 2011. Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting. In *CLOUD*. 500–507.

[44] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. 2011. CloudScale: Elastic Resource Scaling for Multi-tenant Cloud Systems. In *SOCC*. 1–14.

[45] Rebecca Taft, Nosayba El-Sayed, Marco Serafini, Yu Lu, Ashraf Aboulnaga, Michael Stonebraker, Ricardo Mayerhofer, and Francisco Andrade. 2018. P-Store: An Elastic Database System with Predictive Provisioning. In *SIGMOD*. 205–219.

[46] Lalitha Viswanathan, Bikash Chandra, Willis Lang, Karthik Ramachandra, Jignesh M. Patel, Ajay Kalhan, David J. DeWitt, and Alan Halverson. 2017. Predictive Provisioning: Efficiently Anticipating Usage in Azure SQL Database. In *ICDE*. 1111–1116.

PGE: Robust Product Graph Embedding Learning for Error Detection

Kewei Cheng
viviancheng@cs.ucla.edu
University of California, Los Angeles

Xian Li
xianlee@amazon.com
Amazon.com

Yifan Ethan Xu
xuyifa@amazon.com
Amazon.com

Xin Luna Dong
lunadong@gmail.com
Facebook.com

Yizhou Sun
yzsun@cs.ucla.edu
University of California, Los Angeles

ABSTRACT

Although product graphs (PGs) have gained increasing attentions in recent years for their successful applications in product search and recommendations, the extensive power of PGs can be limited by the inevitable involvement of various kinds of errors. Thus, it is critical to validate the correctness of triples in PGs to improve their reliability. Knowledge graph (KG) embedding methods have strong error detection abilities. Yet, existing KG embedding methods may not be directly applicable to a PG due to its distinct characteristics: (1) PG contains rich textual signals, which necessitates a joint exploration of both text information and graph structure; (2) PG contains a large number of attribute triples, in which attribute values are represented by free texts. Since free texts are too flexible to define entities in KGs, traditional way to map entities to their embeddings using ids is no longer appropriate for attribute value representation; (3) Noisy triples in a PG mislead the embedding learning and significantly hurt the performance of error detection. To address the aforementioned challenges, we propose an end-to-end noise-tolerant embedding learning framework, PGE, to jointly leverage both text information and graph structure in PG to learn embeddings for error detection. Experimental results on real-world product graph demonstrate the effectiveness of the proposed framework comparing with the state-of-the-art approaches.

PVLDB Reference Format:
Kewei Cheng, Xian Li, Yifan Ethan Xu, Xin Luna Dong, and Yizhou Sun. PGE: Robust Product Graph Embedding Learning for Error Detection. PVLDB, 15(6): 1288 - 1296, 2022.
doi:10.14778/3514061.3514074

1 INTRODUCTION

With the rapid growth of the internet, e-commerce websites such as Amazon, eBay, and Walmart provide important channels to facilitate online shopping and business transactions. As an effective way to organize product-related information, product knowledge graphs (PGs) [14] have attracted increasing attentions in recent years by empowering many real-world applications, such as product search and recommendations [2].

Proceedings of the VLDB Endowment, Vol. 15, No. 6 ISSN 2150-8097.
doi:10.14778/3514061.3514074

Title	Category	Flavor	Ingredient
Brand A Tortilla Chips Spicy Queso, 6 - 2 oz bags †	chips-and -crisps	Spicy Queso	Ground Corn, Chipotle Pepper Powder, Paprika Extract, Spices
Brand B Bean Chips Spicy Queso, High Protein and Fiber, Gluten Free, Vegan Snack, 5.5 Ounce (Pack of 6) †	chips-and -crisps	Cheddar	Navy Beans, Cayenne Pepper, Paprika Extract, Dehydrated Spices
Carolina Reaper Spicy Peanut Brittle	candy -brittle	Carolina Reaper Spicy	Peanuts, Sugar, Carolina Reaper

† We mask the brand of the products to avoid revealing sensitive information.

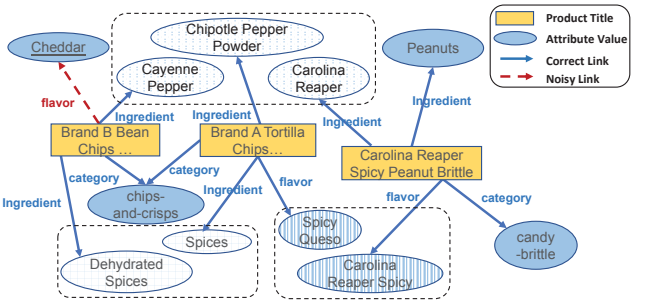


Figure 1: An example PG and its corresponding product catalog data. We underline the incorrect attribute value in the table whose ground truth value is given in its product title. Attribute values with similar semantic meanings are filled with the same pattern and gathering together with a dotted frame.

A PG is a knowledge graph (KG) that describes product attribute values. It is constructed based on product catalog data (Fig. 1 shows an example). In a PG, each product is associated with multiple attributes such as product brand, product category, and other information related to product properties such as flavor and ingredient. Different from traditional KGs, where most triples are in the form of (head entity, relation, tail entity), the majority of the triples in a PG have the form of (product, attribute, attribute value), where the attribute value is a short text, e.g., *("Brand A Tortilla Chips Spicy Queso, 6 - 2 oz bags", flavor, "Spicy Queso")*. We call such triples *attribute triples*.

A vast majority of the product catalog data are provided by individual retailers. These self-reported data inevitably contain many kinds of errors, including conflicting, erroneous, and ambiguous values. When such errors are ingested by a PG, they lead to unsatisfying performance of its downstream applications. Due to the huge

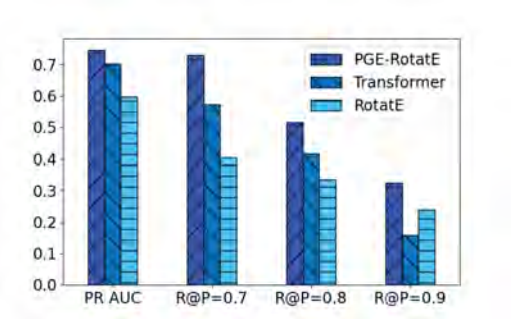


Figure 2: PGE improves over KG embedding method RotatE by 24.7% and transformer by 4% on PR AUC in transductive setting. It also shows significant improvement on R@P metric. R@P = 0.7 shows the recall when the precision is 0.7, etc.

volume of products in a PG, manual validation is not feasible. An automatic validation method is in urgent need.

Knowledge graph embedding (KGE) methods currently hold the state-of-the-art in learning effective representations for multi-relational graph data. It aims to learn the network structure which triples should comply. KG embedding methods have shown promising performances in error detection (i.e., determine whether a triple is correct or not) in KGs [1, 40]. For example, the PG structure in Fig. 1 indicates a strong correlation between the ingredient "pepper" and flavor "spicy" because they are connected through multiple products. By verifying its consistency with the network structure, the errors like *("Brand B Bean Chips Spicy Queso, High Protein and Fiber, Gluten Free, Vegan Snack, 5.5 Ounce (Pack of 6)", flavor, "Cheddar")* can be easily identified. Unfortunately, the existing KG embedding methods cannot be directly used to detect errors in a PG because of the following challenges.

C1: PG contains rich textual information. Products in a PG are often described by short texts like their titles and descriptions that contain rich information about their attributes. For example, the product title "Brand A Tortilla Chips Spicy Queso, 6 - 2 oz bags" covers multiple attributes, including brand, product category, flavor, and size. We can easily verify the correctness of these attributes against the product title. In addition, the attribute values in PG are free texts. Thus the traditional way of mapping entity ids to their embeddings is no longer appropriate. As shown in Fig. 1, when the attribute values "Chipotle Pepper Powder" and "Carolina Reaper" (a kind of pepper) are modeled as two independent entities using their ids, the strong conceptual correlation between the ingredient "pepper" and the flavor "spicy" is lost. Although several recent publications [37, 38] tried to exploit the rich textual information in KGs, the network structure and text information were not jointly encoded into a unified representation. For example, text-based representation and structure-based representation were learned by separate loss functions and integrated into one joint representation by a linear combination [1, 38].

C2: PG contains a large number of unseen attribute values. The flexibility of textual attribute values also makes handling "unseen attribute values" challenging. In the example as shown in Fig. 1, we can learn the representation of "Chipotle Pepper Powder" during training, but an unobserved attribute value with similar semantic meaning, such as "Chipotle Pepper" might be given for validation. Conventional KG embedding models cannot deal with this inductive setting because they have no representations for the entities outside of KGs.

Table 1: Capabilities of different methods.

Methods	Modeling graph structure	Modeling textual data	Noise-aware
Structure based KG embedding [9, 31, 32, 41]	✓		
Text and KG joint embedding [1, 37, 38]	✓	✓	
Noise-aware KG embedding [39]	✓		✓
PGE	✓	✓	✓

C3: Existing noisy data in PG make it hard to learn a reliable embedding model. Getting a reliable embedding model for error detection in a PG requires clean data for training. However, noise widely existing in a PG can mislead the embedding model to learn the wrong structure information, which may severely downgrade its performance in error detection.

No existing approach is capable of tackling all aforementioned challenges, as shown in Table 1. Therefore, in this paper, we aim to answer this challenging research question: *how to generate embeddings for a text-rich, error-prone knowledge graph to facilitate error detection?* We present a novel embedding learning framework, robust **P**roduct **G**raph **E**mbedding (**PGE**), to learn effective embeddings for such knowledge graphs. There are two key underlying ideas for our framework. First, our embeddings seamlessly combine the signals from the textual information of attribute triples, and the structural information in the knowledge graph. We do this by applying a CNN encoder to learn text-based representations for product titles and attribute values, and then integrating these text-based representations into the triplet structure to capture the underlying patterns in the knowledge graph. Second, we present a noise-aware loss function to prevent noisy triples in the PG from misguiding the embeddings during training. For each positive instance in the training data, our model predicts the correctness of the triple according to its consistency with the rest of the triples in the KG, and downweights an instance when the confidence of its correctness is low. As shown in Table 1, PGE is able to model both textual evidence and graph structure, and is robust to noise.

Our proposed model is generic and scalable. First, it applies not only on the product domain, but also excel in other domains such as on Freebase KG, as we show in our experiments. Second, through careful choices of the deep learning models, our model can be trained on KGs with millions of nodes within a few hours, and are robust to noises and unseen values that are inherent in real data. In summary, this paper makes the following contributions.

- We propose an end-to-end noise-tolerant embedding learning framework, PGE, to jointly leverage both text information and graph structure in PG to learn embeddings for error detection.
- We propose a novel noise-aware mechanism to incorporate triple confidence into PGE model to detect noise while learning knowledge representations simultaneously.

- We evaluate PGE on a real-world PG w. millions of nodes generated from public Amazon website and show that we are able to improve over state-of-the-art methods on average by 18% on PR AUC in transductive setting as summarized in Figure 2.

2 PRELIMINARIES AND PROBLEM DEFINITION

We first formally define two important concepts: attribute triples and product graph.

Definition 1. *Attribute triples*

An attribute triple can be represented as (t, a, v), where its subject entity t is a product sold on Amazon (e.g., a product with title "Brand A Tortilla Chips Spicy Queso, 6 - 2 oz bags"), its object entity v is an attribute value (e.g., "spicy queso"), and a is an attribute to connect t and v (e.g., flavor). Both t and v are represented as unstructured short texts. An attribute triple (t, a, v) is *incorrect* if its attribute value v does not correctly describe the product t. For example, *("Brand B Bean Chips Spicy Queso, High Protein and Fiber, Gluten Free, Vegan Snack, 5.5 Ounce (Pack of 6)", flavor, "Cheddar")* in Fig. 1 is an incorrect attribute triple.

Definition 2. *Product Graph*

A Product graph (PG) is a KG that describes product attribute values. Formally, we represent a product graph as $\mathcal{G} = \{T, A, V, O\}$, where T is a set of product titles, A is a set of attributes, V is a set of product attribute values, and O is a set of observed triples in the PG. Note that we have open-world assumption and thus cannot predetermine the possible values of V. Triples in PG are attribute triples defined in Definition 1. Fig. 1 illustrates an example PG.

We can now formally define the problem of *error detection in PG* as follows:

Given: a product graph $\mathcal{G} = \{T, V, A, O\}$.
Identify: incorrect triples $\{(t, a, v)\} \subset O$.

3 OUR PROPOSED FRAMEWORK: PGE

In this section, we present PGE that learns the embeddings of PG entities by incorporating both the text information and the network structure of a PG to detect erroneous triples. As shown in Fig. 3, the framework includes three key components: (1) Learn text-based representations of entities from their raw text values; (2) Leverage network structure of a PG to guide the final embedding learning for error detection; (3) Introduce a noise-aware mechanism to diminish the impact of noisy triples to the representation learning.

3.1 Text-based Representation Learning for Entities

In a typical KG embedding learning procedure, each entity is given a unique id which is then mapped to a learnable embedding. This approach is not optimal for PG embedding learning, because product titles (T) and attribute values (V) in a PG are mostly unstructured text containing rich semantic information, thus learning entity embeddings from only their ids not only creates unnecessary degrees of freedom, but also discards their underlying semantic connections. For instance, the embeddings of product titles "Brand A Tortilla Chips Spicy Queso, 6 - 2 oz bags" and "Brand B Bean Chips Spicy Queso, High Protein and Fiber, Gluten Free, Vegan Snack, 5.5 Ounce (Pack of 6)" should be close to each other because they are semantically similar. There are several methods, such as convolutional neural network (CNN)-based methods and Transformer-based methods (e.g., BERT), that could be leveraged to learn the representations of product titles (T) and attribute values (V) in order to capture their semantic similarities. We present scalability analysis of both text encoders in Section 4.6. Due to the huge number of products contained in PGs, we pick the CNN architecture for its good scalability as well as effectiveness on many natural language processing tasks [28]. As shown in Figure 4, the CNN encoder takes the raw text of a product title or an attribute value as the input and output its text-based representation. The first layer in the encoder transforms every word in the sequence into its respective embedding (initialized with word2vec [25]). The word embeddings then pass through three 1-d shallow CNNs with different filter sizes, which create three feature maps. Here we use different filter sizes to capture local semantic information from different text spans. The final text-based representation of an entity is the concatenated feature maps learned by all CNNs.

3.2 Leverage Graph Structure to Guide Embedding Learning

Manually labeled data are costly to obtain given the huge number of products in a PG. Fortunately, the rich structure information of a PG bridges the gap between the difficulties in obtaining labeled data and the necessity of supervision to detect errors. Although several recent papers have proposed to combine the text and structure information for KG representation learning, most of them [1, 38] learn two independent representations with separate loss functions and then integrate them with a linear combination. Such solution cannot generate a desired unified representation. To address this issue, we propose to learn the embeddings of entities and relations end to end, encoding the network structure that triples should obey on top of their text-based representations.

As shown in Fig. 3, we introduce a fully-connected neural network layer to transform a text-based representation into its final representation to encode the network structure of a PG. Boldfaced $\mathbf{t}$, $\mathbf{a}$, $\mathbf{v}$ denote the final embedding vector of product title t, attribute a, attribute value v, respectively. Since the number of attributes in a PG is small and well-defined comparing to titles and attribute values, we use randomly initialized learnable vectors to represent relations instead of CNN encoders. To capture the network structure of PG, we define the objective function by maximizing the joint probability of the observed triples given the embeddings of both entities and relations. In particular, we assume all triples are conditionally independent given the corresponding embeddings. Then the joint distribution of all the triples is defined as:

$$P(O) = \prod_{(t,a,v) \in O} P\Big((t, a, v) | \{\mathbf{t}\}, \{\mathbf{a}\}, \{\mathbf{v}\}\Big). \tag{1}$$

Since our goal is to detect the incorrect attribute value v in a triple (t, a, v), we optimize $P(v|t, a, \{\mathbf{t}\}, \{\mathbf{a}\}, \{\mathbf{v}\})$ instead of $P(t, a, v)|\{\mathbf{t}\}, \{\mathbf{a}\}, \{\mathbf{v}\})$, which can be formalized as follows:

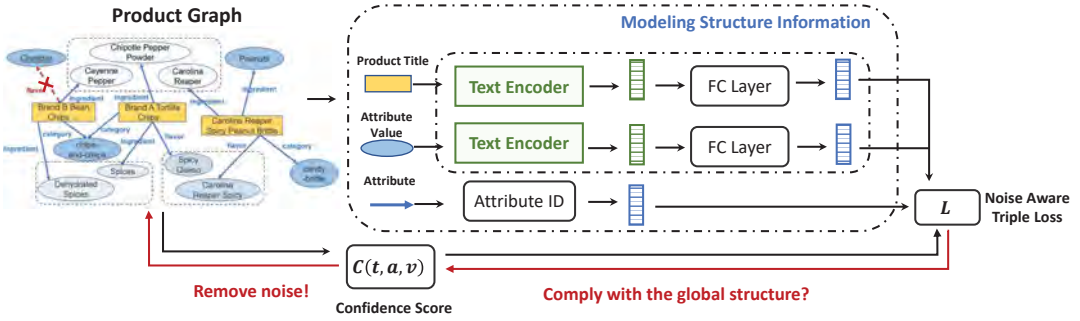


Figure 3: Illustration of the end-to-end PGE framework. The embedding vectors in green are text-based entities representations learned from text descriptions, while the embedding vectors in blue are the final entity embeddings learned under the guidance of the PG network structure. The arrows in red illustrate how the noise-aware mechanism removes noises in PG.

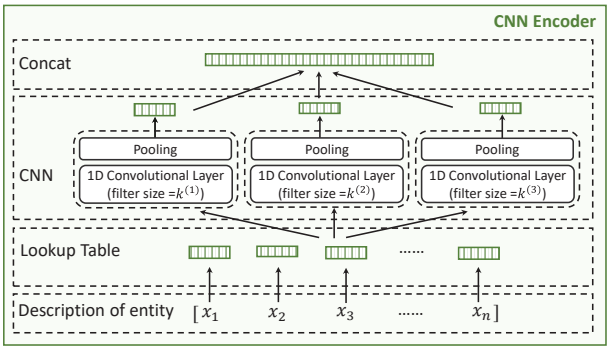


Figure 4: CNN-based text encoder.

$$P(v|t,a,\{\mathbf{t}\},\{\mathbf{a}\},\{\mathbf{v}\}) = \frac{\exp\Big(f_a(t,v)\Big)}{\sum_{v' \in V} \exp\Big(f_a(t,v')\Big)} \tag{2}$$

where $f_a(t,v)$ can be defined by any KG embedding scoring functions. For example, in TransE, $f_a(t,v) = \gamma - \|\mathbf{t}+\mathbf{a}-\mathbf{v}\|_1^2$, where $\mathbf{t},\mathbf{a},\mathbf{v} \in \mathbb{R}^d$ and γ is a fixed margin. In particular, a higher $f_a(t,v)$ usually indicates that the triple (t,a,v) is more plausible. Due to the large number of attribute values $|V|$ involved in a PG, it is impractical to directly compute the softmax functions. Therefore, we adopt negative sampling [26] as computationally efficient approximation instead and reformulate the objective function as follows:

$$\begin{aligned} &\sum_{(t,a,v)\in O} \Big[-\log\sigma\Big(f_a(t,v)\Big) \\ &-\frac{1}{|\mathcal{N}(t,a,v)|} \sum_{(t,a,v')\in\mathcal{N}(t,a,v)} \log\sigma\Big(-f_a(t,v')\Big)\Big] \end{aligned} \tag{3}$$

where σ is the standard sigmoid function, O represents the observed facts in PG, $\mathcal{N}(t,a,v)$ is a set of negative samples for an attribute triple (t,a,v). More specifically, for each observed triple (t,a,v) we sample a set of negative samples $\mathcal{N}(t,a,v) \subset \{(t,a,v')|v' \in V\}$ by replacing the attribute value v with a random value from V.

3.3 Noise-aware Mechanism

The objective function in Eq. (3) indiscriminately minimizes the scores of all facts in PG without taking their trustworthiness into consideration. As a result, noisy facts can mislead the embedding model to learn wrong structure information, thus harm the performance of embeddings in error detection.

To address this issue, we propose a novel noise-aware mechanism to reduce the impact of noisy triples on the representation learning process. Knowledge representations are learned to ensure global consistency with all triples in PG. Correct triples are inherently consistent, which can jointly represent the global network structure of PG; noisy triples usually conflict with these global network structures. Consequently, by forcing consistency between correct triples and noises, performance is unnecessarily sacrificed. The main idea of the noise-aware mechanism is to explicitly allow the model to identify and "markdown" a small set of incorrect triples during training and reduce their impact on the loss function.

More specifically, we introduce a binary learnable confidence score, $C(t,a,v)$, for every triple (t,a,v) in a PG to indicate whether the fact is true or false. $C(t,a,v) = 1$ indicates the triple is correct and 0 otherwise. Associating confidence scores with triples in a PG actively downweight potential noises in the PG. The objective function of the noise-aware PGE model is defined as follows.

$$\begin{aligned} \mathcal{L} = &\sum_{(t,a,v)\in O} C(t,a,v)\Big[-\log\sigma\Big(f_a(t,v)\Big) \\ &-\frac{1}{|\mathcal{N}(t,a,v)|} \sum_{(t,a,v')\in\mathcal{N}(t,a,v)} \log\sigma\Big(-f_a(t,v')\Big)\Big] \\ &+\alpha \sum_{(t,a,v)\in O} \Big(1-C(t,a,v)\Big) \\ &s.t., C(t,a,v) \in \{0,1\} \end{aligned} \tag{4}$$

where $C(t,a,v)$ is the binary confidence score assigned to a triple (t,a,v) in a PG, and $\alpha \sum_{(t,a,v)\in O} \Big(1-C(t,a,v)\Big)$ is a regularization term imposed on confidence scores to control their sparsity. The problem in Eq.(4) is difficult to solve due to the boolean constraint on $C(t,a,v)$. Following the common relaxation technique in [34],

the boolean constraint on $C(t,a,v)$ can be relaxed as:

$$C(t,a,v)^2 + \Big(1 - C(t,a,v)\Big)^2 = 1, \tag{5}$$

since minimizing $1 - C(t,a,v)^2 - \Big(1 - C(t,a,v)\Big)^2$ polarizes $C(t,a,v)$. Therefore, we rewrite the Eq. (4) as:

$$\begin{aligned}\mathcal{L} = \sum_{(t,a,v)\in O} C(t,a,v)\Big[&- \log\sigma\Big(f_a(t,v)\Big) \\ &- \frac{1}{|\mathcal{N}(t,a,v)|}\sum_{(t,a,v')\in\mathcal{N}(t,a,v)} \log\sigma\Big(-f_a(t,v')\Big)\Big] \\ &+ \alpha \sum_{(t,a,v)\in O}\Big(1 - C(t,a,v)\Big) \\ &+ \beta \sum_{(t,a,v)\in O}\Big(1 - C(t,a,v)^2 - \Big(1 - C(t,a,v)\Big)^2\Big).\end{aligned} \tag{6}$$

4 EXPERIMENTS

4.1 Dataset

We evaluate our PGE on two datasets: one real-world e-commerce dataset collected from publicly available Amazon webpages, and one widely used benchmark dataset FB15K-237. Table 2 summarizes the statistics of both datasets.

Amazon Dataset: To evaluate PGE on real-world e-commerce dataset, we construct a product graph with the product data obtained from public Amazon website. Each product in the Amazon dataset is associated with multiple attributes, such as product title, brand and flavor, whose values are short texts. As shown in Table 2, the Amazon dataset contains 750,000 products associated with 27 structured attributes and 5 million triples. To avoid bias, we sampled products from 325 product categories across different domains, such as food, beauty and drug. To prepare labeled test data, we asked Amazon Mechanical Turk (MTurk) workers to manually label the correctness of two attributes, including *flavor* and *scent*, based on corresponding product profiles. Each data point is annotated by three Amazon Mechanical Turk workers and the final label is decided by majority voting. Among 5,782 test triples, 2,930 are labeled as incorrect and 3,304 are labeled as correct.

FB15K-237: The FB15K dataset is the most commonly used benchmark knowledge graph dataset [9]. It contains knowledge graph relation triples and textual mentions of Freebase entity pairs. FB15K-237 is a variant of FB15K dataset where inverse relations are removed to avoid information leakage problem in test dataset. The FB15K-237 datasets benefit from human curation that results in highly reliable facts. We add 10% noisy triples to the data set by randomly sample 10% triples and substituting the original head or tail entity with a randomly selected entity.

4.2 Experimental Setting

Our goal is to identify incorrect attribute values of a product, which can be formally defined as a triple classification problem in PG. We choose a threshold θ based on the best classification accuracies on the validation dataset, then classify a triple (t,a,v) as correct if its score $f_a(t,v) > \theta$, otherwise incorrect. We apply the same settings to all baseline methods to ensure a fair comparison. We evaluate two versions of our model by incorporating TransE [9] and RotatE [31] as the score function, respectively.

Evaluation Metric. We adopt the area under the Precision-Recall curve (PR AUC) and Recall at Precision=X (R@P=X) to evaluate the performance of the models. To be more specific, PR AUC is defined as the area under the precision-recall curve, which is widely used to evaluate the ranked retrieval results. R@P is defined as the recall value at a given precision, which aims to evaluate the model performance when a specific precision requirement needs to be satisfied. For example, R@R = 0.7 shows the recall when the precision is 0.7.

Compared Methods. We evaluate PGE against state-of-the-art (SOTA) algorithms, including (1) NLP-based method (LSTM, Transformer [33]); (2) structure based KG embedding (TransE [9], DistMult [41], ComplEx [32], RotatE [31]); (3) text and KG joint embedding (e.g., DKRL [38], SSP [37]); and (4) noise-aware KG embedding (CKRL [39]). We choose CNN and BERT as the text encoders of PGE. Since BERT cannot handle Amazon dataset due to scalability issues, only the results of CNN is reported in Section 4.3 and Section 4.4. We present scalability analysis of both text encoders in Section 4.6. We also include the approach "Union of Transformer and PGE" to show how PGE complement Transformer. To combine Transformer and PGE for error detection, the approach "Union of Transformer and PGE" re-ranks the test triples by jointly considering the ranking given by the Transformer and PGE. For example, given a test triple (h,r,t), suppose Transformer rank it as i while PGE rank it as j. Then the average ranking of triple (h,r,t) is $R_{avg}^{(h,r,t)} = (1/i + 1/j)/2$. Based on $R_{avg}^{(h,r,t)}$, "Union of Transformer and PGE" re-ranks the test triples. Smaller $R_{avg}^{(h,r,t)}$ results in higher ranking assigned by "Union of Transformer and PGE". In addition to "Union of Transformer and PGE", we also include a strong ensemble method - RotatE+ to enrich knowledge graph with information extraction technique. In particular, RotatE+ first applies OpenTag [20, 43], the SOTA information extraction toolkit developed by Amazon Product Graph Team, to extract all relevant attributes from product title and product description to enrich the PG, then applies KG embedding method RotatE on the enriched KG to detect the error.

Setup Details. In data preprocessing, we remove all stop words from raw texts and map words to 300-dimensional word2vec vectors trained with GoogleNews. We adopt the Adam [23] optimizer with learning rate among $\{0.0001, 0.0002, 0.0005\}$ following [31], and margin γ among $\{12.0, 24.0\}$. For the CNN encoder, we try different filter sizes among $\{1,2,3,4\}$ for different CNNs. To fairly compare with different baseline methods, we set the parameters for all baseline methods by a grid search strategy. The best results of baseline methods are used to compare with PGE.

4.3 Transductive Setting

Transductive setting focuses on the situation where all attribute values in the test triples have been observed in the training stage. To compare different algorithms on the triple classification task, we require each method to predict the correctness of triples in the test dataset. Table 3 shows the comparison results. Here are several interesting observations: (1) PGE consistently outperforms KG embedding models as well as CKRL in all cases with significant

Table 2: Data statistics

Dataset	#Relations	#Entities	#Products	#Attributed values	#Train	#Valid	#Test
Amazon Dataset	27	1,017,374	750,000	267,374	4,989,375	6,924	5,782
FB15K-237	234	13,714	-	-	67,894	2,750	3,042

performance gain (improving by 24% - 30% on PR AUC), which ascribes to the utilization of textual information associated with entities; (2) PGE also obtains better performance than NLP-based approaches as they cannot leverage graph structure information in KGs. In particular, NLP-based methods show the worst performance on the FB15k-237 dataset while the second best performance on Amazon dataset. The major reason is that FB15k-237 contains much richer graph information compared to the Amazon dataset (i.e., there are 27 attributes in Amazon dataset while 234 relations in FB15k-237). Therefore, graph structure plays a more critical role in error detection task in FB15k-237; (3) PGE shows better performance compared to DKRL and SSP. The major reason is that DKRL and SSP learn the structural representations and the textual representations by separate functions.

4.4 Inductive Setting

Inductive setting focuses on the situation where attribute values in the test triples are not presented in a PG, which is a common scenario for PG error detection. Existing KG embedding models are not effective in dealing with this situation because they cannot generate representations for the entities outside of KGs due to missing ids. Therefore, we do not include them as baselines in this subsection. Unlike the KG embedding methods, which map entities to their embeddings using ids, our proposed PGE learns embeddings of entities based on their text-based representations and thus can naturally handle the inductive setting.

To prepare an inductive setting, we filter the training set by excluding any triples that share entities with the selected test triples, so that the training and the testing use disjoint sets of entities. We report the results on R@P=0.6, R@P=0.7, R@P=0.8 in Table 4. We observe that: (1) All methods perform worse in the inductive setting without exception, which indicates that inductive setting is indeed more challenging; (2) NLP-based methods perform the best among all methods. The major reason is that language naturally has strong transferring ability while PGE still relies on the graph structure to make the prediction. Although text encode can transferring information among entities, it doesn't help to predict a never seen graph structure; (3) Although NLP-based methods perform better than PGE on the Amazon dataset, the best results are given by the union of Transformer and PGE (improving by 9% on R@P=0.9 compared with Transformer), showing that PGE can learn the undetected error by Transformer ; (4) Although PGE cannot leverage textual information as well as Transformer (because CNN is less powerful compared with Transformer in capturing the semantic information. Not to mention we employ shallow CNN as text encode due to scalability issues), it still achieves comparable result on the Amazon dataset. Moreover, they achieve the SOTA on FB15k-237, which further validates the strong ability of PGE in detecting errors in a KG with rich textual information; (5) DKRL and SSP perform the worst among all methods, which again demonstrates their weakness.

4.5 Validity of Noise-aware Mechanism

Validity of Confidence Scores with Different Injected Noises. To evaluate the benefit of including confidence scores $C(t, a, v)$ in the noise-aware mechanism, shown in Eq. (6) , we evaluate *PGE(CNN)-RotatE* on the Amazon dataset with two different kinds of injected noises. First, we inject human-labeled correct triples and incorrect triples into the training data. Confidence scores are learned to determine the correctness of these injected labeled triples. The distribution of confidence scores are shown in Fig. 5 (a). Second, we inject artificial noises into the training data. We substitute attribute values of existing triples in the Amazon dataset with a random value to generate these artificial noises. Confidence scores are learned to distinguish artificial noises from the original triples. Fig. 5 (b) shows the distribution of confidence scores. The red bars represent the distribution of confidence scores for human-labeled incorrect triples (or injected artificial noises) while the blue bars represent the distribution of confidence scores for human-labeled correct triples (or triples in the original Amazon dataset). We observe that real-world noises are more difficult to identify compared to artificial noises. Despite the difficulty in detecting the real-world error, confidence scores of human-labeled correct triples are mainly over 0.5, validating the promising capability of the confidence scores to distinguish noises in PG. In addition, we observe that 1% triples in the original Amazon dataset have also been identified as noises in Fig. 5 (b). We have verified that most of these triples are indeed noisy triples in the original Amazon dataset.

Overall Impact of Noise-aware Mechanism. To further validate the overall benefits brought by noise-aware mechanism, we also evaluate *PGE(CNN)-RotatE* without noise-aware mechanism on Amazon dataset used in Section 4.3. Figure 6 presents the comparison results. We observe that the noise-aware mechanism brings significant performance gain: PGE(CNN)-RotatE with noise-aware mechanism increases the PR AUC of PGE(CNN)-RotatE without noise-aware mechanism from 0.734 to 0.747 and increases R@P=0.9 from 0.289 to 0.325.

4.6 Scalability Analysis

To demonstrate the scalability of PGE, we present the training time of PGE on Amazon dataset of different sizes in Table 5. We vary the sample ratio among $\{0.1, 0.3, 0.5, 0.7, 1\}$ to select only a portion of triples in Amazon dataset to construct PG of different sizes. Two text encoders, CNN-based text encoder and BERT-based text encoder, are leveraged to learn entity representations. In particular, BERT-based text encoder takes the raw text of product titles or attribute values as input. The first token of input is always a special classification token ([CLS]). The final hidden state corresponding to this token is used as the text-based representation of entities. We observe that *PGE(BERT)-RotatE* cannot be applied to Amazon dataset due to the scalability issue. It takes near 2 days for 10% data and over 3 days for 30% data. Therefore, we focus on CNN in this

Table 3: Results of error detection under the transductive setting. The numbers in bold represent the best performance among all methods while the numbers underlined represent the second best. Evaluation of PGE on the Amazon dataset shows that PGE is able to improve over the SOTA methods on average by 18% on PR AUC.

Categories	Method	Amazon Dataset					FB15k-237				
		PR AUC	R@P=0.7	R@P=0.8	R@P=0.9	Time (hours)	PR AUC	R@P=0.7	R@P=0.8	R@P=0.9	Time (hours)
NLP-based methods	LSTM	0.704	0.572	0.416	0.159	16.32	0.626	0.595	0.445	0.239	0.43
	Transformer [33]	0.719	0.601	0.427	0.194	79.46	0.648	0.649	0.503	0.245	12.82
Structured based KG embedding	TransE [9]	0.584	0.390	0.308	0.213	20.57	0.772	0.793	0.737	0.685	0.58
	DistMult [41]	0.573	0.362	0.291	0.197	32.86	0.819	0.872	0.813	0.751	4.12
	ComplEx [32]	0.579	0.373	0.310	0.207	36.31	0.781	0.814	0.759	0.712	5.16
	RotatE [31]	0.597	0.405	0.336	0.239	35.11	0.824	0.875	0.823	0.766	5.33
	RotatE+‡	0.611	0.423	0.369	0.221	36.79	-	-	-	-	-
Text and KG joint embedding	DKRL [38]	0.693	0.552	0.408	0.246	45.38	0.909	0.945	0.901	0.868	7.25
	SSP [37]†	-	-	-	-	-	0.927	0.951	0.915	0.882	-
Noise-aware KG embedding	CKRL [39]	0.586	0.392	0.304	0.217	21.16	0.768	0.725	0.672	0.627	0.62
Our Proposed model	PGE(CNN)-TransE	0.738	0.690	0.436	0.267	23.12	**0.990**	**0.997**	**0.995**	**0.986**	0.67
	PGE(CNN)-RotatE	0.745	0.729	**0.516**	0.325	39.41	**0.990**	**0.997**	0.993	0.983	5.71
Union of Transformer and PGE(CNN)-RotatE		**0.751**	**0.747**	0.509	**0.349**	-	0.938	0.958	0.911	0.893	-

† Since SSP cannot handle Amazon dataset due to scalability issues, only the results on the FB15K-237 is reported.

‡ RotatE+ first applies OpenTag [20, 43], an information extraction toolkit developed by Amazon Product Graph Team, to extract all relevant attributes from product title and product description to enrich the product graph, then apply RotatE [31] on the enriched KG to detect the error.

Table 4: Results of error detection under the inductive setting. The bold numbers represent the best performances among all methods while the underlined numbers represent the second best. We observe that PGE achieves the SOTA on the structure-rich FB15k-237 data set. The best results on the Amazon dataset are given by the union of the Transformer model and PGE, showing that although PGE does not perform as well as NLP-based methods on the Amazon dataset, it complements Transformer for its strong ability in capturing graph structure.

Categories	Method	Amazon Dataset				FB15k-237			
		PR AUC	R@P=0.6	R@P=0.7	R@P=0.8	PR AUC	R@P=0.6	R@P=0.7	R@P=0.8
NLP-based methods	LSTM	0.626	0.756	0.476	0.340	0.581	0.717	0.436	0.204
	Transformer [33]	0.643	0.771	0.495	0.354	0.603	0.748	0.453	0.238
Text and KG joint embedding	DKRL [38]	0.552	0.593	0.252	0.068	0.698	0.790	0.638	0.415
	SSP [37]†	-	-	-	-	0.716	0.807	0.654	0.419
Our Proposed model	PGE(CNN)-TransE	0.585	0.730	0.412	0.197	0.787	0.871	0.724	0.674
	PGE(CNN)-RotatE	0.596	0.741	0.437	0.228	**0.836**	0.919	**0.845**	**0.753**
Union of Transformer and PGE(CNN)-RotatE		**0.649**	**0.779**	**0.512**	**0.386**	0.833	**0.923**	0.837	0.743

† Since SSP cannot handle Amazon dataset due to scalability issues, only the results on the FB15K-237 is reported.

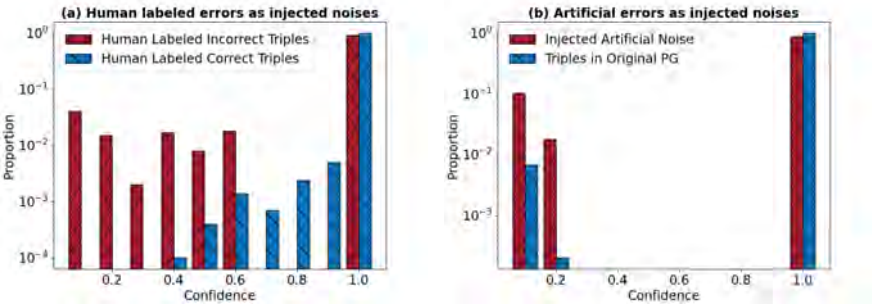


Figure 5: Distribution of confidence scores learned by PGE(CNN)-RotatE on the Amazon dataset with different injected noises.

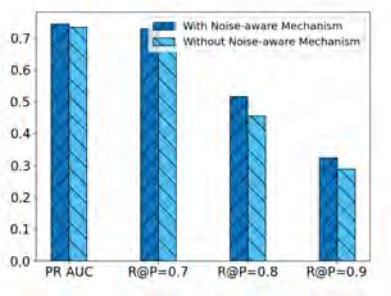


Figure 6: PGE(CNN)-RotatE with v.s. without noise-aware mechanism on noisy Amazon dataset.

paper. We observe that *PGE(CNN)-RotatE* scales up to large datasets with similar scalability compared to KGE model.

4.7 Case Study

Previous experiments have shown the promising performance of PGE in both transductive setting and inductive setting. To further demonstrate the capability of PGE in detecting real-world errors in PG, we conduct case study to give examples of identified errors in the Amazon dataset as shown in Table 6. We use PGE(CNN)-RotatE to evaluate if a triple is a correct fact. Threshold σ is chosen based on the best classification accuracies on the validation dataset in order to classify triples. We observe that most attribute values of identified errors violate the global graph structure of PG and thus can be classified as errors. For example, product 2,3, and 4 in Table 6 are not groceries thus should not have the attribute "flavor".

Table 5: Training Time (hours) of different methods on Amazon dataset.

Models	Percentage of Sampled Triples				
	0.1	0.3	0.5	0.7	1
RotatE	3.22	10.77	17.63	24.86	35.11
PGE(CNN)-RotatE	4.07	11.95	19.44	27.62	39.41
PGE(BERT)-RotatE	45.21	> 3 day	> 3 day	> 3 day	> 3 day

Table 6: Identified errors on Amazon dataset.

Product	Attribute	Attribute Value
Pure Mint Shampoo and Hair Conditioner for Women and Men - 10 oz	scent	mint shampoo and conditioner set
Brand A Foot Brush and Pumice (Pack of 4) †	flavor	bamboo
Brand B Sweet BBQ Rub 11.2 oz†	flavor	sweet
Hassle Free Storage Pop-Up Mesh Laundry Hamper (Aqua)	flavor	octopus
Brand C Organics Conditioner, Tea Tree Oil & Blue Cypress, † 12 Ounce (Pack of 3)	scent	conditioner tea tree oil and blue cypress

† We mask the brand of the products to avoid revealing sensitive information.

Although the attribute values of product 1 and 5 include commonly observed phrases to describe the scent, the word "conditioner" in the attribute values makes them no longer correct attribute values of "scent". This observation shows that PGE not only leverages the graph structure of PG to detect noise (e.g., example 2,3,4) but also show sensitivity to subtle differences of language.

5 RELATED WORK

Error Detection in Knowledge Graph. Most KG noise detection process is carried out when constructing KGs, such as Freebase, Google Knowledge Graph, Walmart product graph, YAGO, NELL, and Wikipedia [3, 8, 19, 27, 30]. Despite the efforts during KG constructions, errors are widely observed in existing KGs. A recent open IE model on the benchmark achieves only 24% precision when the recall is 67% [29] and the estimated precision of NELL is only 74% [10]. To detect errors for an existing KG, most existing methods explore additional rules [5–7, 11, 12, 15–18, 22]. Considering all kinds of errors that could be made in the real world, it is unrealistic to identify all required rules to cover all possible cases. In contrast, our proposed method employs KG embedding model to automatically learn the correlation of entities, which could be considered as fuzzy rules to guide value cleaning in KGs. More recently, detecting noises while learning knowledge representations simultaneously becomes a hot topic. A confidence-aware framework CKRL [39] is proposed to incorporate triple confidence into KG embedding models to learn noise-aware KG representations. However, the confidence of triples are easily affected by model bias (i.e., improper order of triples in training sets may even amplify the impact of noises). In addition, it ignores the rich semantic information in KGs, which is strong evidence to judge triple quality. In this paper, we propose a noise-aware KG embedding learning method, which can utilize rich semantic information to identify noises, which conflict with the global network structures.

Knowledge Graph Embedding. Knowledge Graph Embedding (KGE) aims to capture the similarity of entities by projecting entities and relations into continuous low-dimensional vectors. Scoring functions, which measure the plausibility of triples in KGs, are the crux of KGE models. Representative KGE algorithms include TransE [9], TransH [36], TransR [24], DistMult [41], ComplEx [32], SimplE [21] and RotatE [31], which differ from each other with different scoring functions.

Text and Knowledge Graph Joint Embeddings. In recent years, several attempts have been made to improve the knowledge representation by exploiting entity descriptions as additional information [1, 37, 40]. However, the combination of the structural and textual representations is not well studied in these methods, in which two representations are learned by separate loss function or aligned only on the word-level. As one of the most representative works, DKRL [38] separates the objective function into two energy functions (i.e. one for structure and one for description) and integrates these two representations into a joint one by a linear combination. Works proposed in [36] and [44] align the entity name with its Wikipedia anchor on word level, which may lose some semantic information on the phrase or sentence level. SSP [37] requires the topic model to learn pre-trained semantic vector of entities separately. Due to the rapid growth of pre-trained language representation models (PLM), several works are proposed to encode textual entity descriptions with a PLM as their embeddings. For example, KEPLER [35] proposes to encode textual entity descriptions with BERT as their embeddings, and then jointly optimize the KGE and language modeling objectives. BLP [13] trains PLM and KG in an end-to-end manner. Since the language modeling objective of PLM suffer from high computational cost and require a large corpus for training, it is time consuming to apply these methods to large scale KGs. In this paper, we propose an end-to-end method to jointly leverage both text information and graph structure for KG embedding learning in an efficient way.

6 CONCLUSION

In this paper, we propose a novel end-to-end noise-aware embedding learning framework, PGE, to learn embeddings on top of text-based representations of entities for error detection in PG. Experiment results on a real-world product graph show that PGE improves over state-of-the-art methods on average by 18% on PR AUC in transductive setting. Although this paper focuses on the product domain, we also show in our experiments that, the same techniques excel in other domains with textual information and noises. As the next step, we would investigate more efficient Transformer architecture to improve text encoder strength and efficiency of PGE. BERT-based text encoder is difficult to scale to large KG due to its full attention mechanism. To reduce the computation complexity of BERT-based text encoder, we can to extend the ideas of [4, 42] to allow sparse self-attention to tokens. In addition, we can leverage additional information to improve the learned entity representations. For example, we could better capture the similarity among products by leveraging the hierarchical structure of product data or by leveraging the user behavior data.

REFERENCES

[1] Bo An, Bo Chen, Xianpei Han, and Le Sun. 2018. Accurate text-enhanced knowledge graph representation learning. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*. 745–755.

[2] Vito Walter Anelli, Pierpaolo Basile, Derek Bridge, Tommaso Di Noia, Pasquale Lops, Cataldo Musto, Fedelucio Narducci, and Markus Zanker. 2018. Knowledge-aware and conversational recommender systems. In *Proceedings of the 12th ACM Conference on Recommender Systems*. 521–522.

[3] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. 2007. Dbpedia: A nucleus for a web of open data. In *The semantic web*. Springer, 722–735.

[4] Iz Beltagy, Matthew E Peters, and Arman Cohan. 2020. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150* (2020).

[5] George Beskales, Ihab F Ilyas, and Lukasz Golab. 2010. Sampling the repairs of functional dependency violations under hard constraints. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 197–207.

[6] George Beskales, Ihab F Ilyas, Lukasz Golab, and Artur Galiullin. 2013. On the relative trust between inconsistent data and inaccurate constraints. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 541–552.

[7] Philip Bohannon, Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. 2007. Conditional functional dependencies for data cleaning. In *2007 IEEE 23rd international conference on data engineering*. IEEE, 746–755.

[8] Kurt Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. 2008. Freebase: a collaboratively created graph database for structuring human knowledge. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM, 1247–1250.

[9] Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. 2013. Translating embeddings for modeling multi-relational data. In *Advances in neural information processing systems*. 2787–2795.

[10] Andrew Carlson, Justin Betteridge, Bryan Kisiel, Burr Settles, Estevam Hruschka, and Tom Mitchell. 2010. Toward an architecture for never-ending language learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 24.

[11] Xu Chu, Ihab F Ilyas, and Paolo Papotti. 2013. Holistic data cleaning: Putting violations into context. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 458–469.

[12] Alvaro Cortés-Calabuig and Jan Paredaens. 2012. Semantics of Constraints in RDFS.. In *AMW*. Citeseer, 75–90.

[13] Daniel Daza, Michael Cochez, and Paul Groth. 2021. Inductive Entity Representations from Text via Link Prediction. In *Proceedings of the Web Conference 2021*. 798–808.

[14] Xin Luna Dong, Xiang He, Andrey Kan, Xian Li, Yan Liang, Jun Ma, Yifan Ethan Xu, Chenwei Zhang, Tong Zhao, Gabriel Blanco Saldana, et al. 2020. AutoKnow: Self-Driving Knowledge Collection for Products of Thousands of Types. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2724–2734.

[15] Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. 2008. Conditional functional dependencies for capturing data inconsistencies. *ACM Transactions on Database Systems (TODS)* 33, 2 (2008), 1–48.

[16] Wenfei Fan, Yinghui Wu, and Jingbo Xu. 2016. Functional dependencies for graphs. In *Proceedings of the 2016 International Conference on Management of Data*. 1843–1857.

[17] Floris Geerts, Giansalvatore Mecca, Paolo Papotti, and Donatello Santoro. 2013. The LLUNATIC data-cleaning framework. *Proceedings of the VLDB Endowment* 6, 9 (2013), 625–636.

[18] Alireza Heidari, Joshua McGrath, Ihab F Ilyas, and Theodoros Rekatsinas. 2019. Holodetect: Few-shot learning for error detection. In *Proceedings of the 2019 International Conference on Management of Data*. 829–846.

[19] Stefan Heindorf, Martin Potthast, Benno Stein, and Gregor Engels. 2016. Vandalism detection in wikidata. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. 327–336.

[20] Giannis Karamanolakis, Jun Ma, and Xin Luna Dong. 2020. Txtract: Taxonomy-aware knowledge extraction for thousands of product categories. *arXiv preprint arXiv:2004.13852* (2020).

[21] Seyed Mehran Kazemi and David Poole. 2018. Simple embedding for link prediction in knowledge graphs. In *Advances in Neural Information Processing Systems*. 4284–4295.

[22] Zuhair Khayyat, Ihab F Ilyas, Alekh Jindal, Samuel Madden, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang, and Si Yin. 2015. Bigdansing: A system for big data cleansing. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 1215–1230.

[23] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[24] Yankai Lin, Zhiyuan Liu, Maosong Sun, Yang Liu, and Xuan Zhu. 2015. Learning entity and relation embeddings for knowledge graph completion. In *Twenty-ninth AAAI conference on artificial intelligence*.

[25] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[26] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*. 3111–3119.

[27] Heiko Paulheim. 2017. Knowledge graph refinement: A survey of approaches and evaluation methods. *Semantic web* 8, 3 (2017), 489–508.

[28] A Rakhlin. 2016. Convolutional Neural Networks for Sentence Classification. *GitHub* (2016).

[29] Gabriel Stanovsky, Julian Michael, Luke Zettlemoyer, and Ido Dagan. 2018. Supervised open information extraction. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*. 885–895.

[30] Fabian M Suchanek, Gjergji Kasneci, and Gerhard Weikum. 2007. YAGO: a core of semantic knowledge. In *Proceedings of the 16th international conference on World Wide Web*. ACM, 697–706.

[31] Zhiqing Sun, Zhi-Hong Deng, Jian-Yun Nie, and Jian Tang. 2019. Rotate: Knowledge graph embedding by relational rotation in complex space. *arXiv preprint arXiv:1902.10197* (2019).

[32] Théo Trouillon, Johannes Welbl, Sebastian Riedel, Éric Gaussier, and Guillaume Bouchard. 2016. Complex embeddings for simple link prediction. In *International Conference on Machine Learning*. 2071–2080.

[33] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

[34] Ulrike Von Luxburg. 2007. A tutorial on spectral clustering. *Statistics and computing* 17, 4 (2007), 395–416.

[35] X Wang, T Gao, Z Zhu, Z Liu, J Li, and J Tang. [n.d.]. KEPLER: A Unified Model for Knowledge Embedding and Pre-trained Language Representation. arXiv 2019. *arXiv preprint arXiv:1911.06136* ([n. d.]).

[36] Zhen Wang, Jianwen Zhang, Jianlin Feng, and Zheng Chen. 2014. Knowledge graph embedding by translating on hyperplanes. In *Twenty-Eighth AAAI conference on artificial intelligence*.

[37] Han Xiao, Minlie Huang, Lian Meng, and Xiaoyan Zhu. 2017. SSP: semantic space projection for knowledge graph embedding with text descriptions. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 31.

[38] Ruobing Xie, Zhiyuan Liu, Jia Jia, Huanbo Luan, and Maosong Sun. 2016. Representation learning of knowledge graphs with entity descriptions. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 30.

[39] Ruobing Xie, Zhiyuan Liu, Fen Lin, and Leyu Lin. 2018. Does William Shakespeare really write Hamlet? knowledge representation learning with confidence. In *Thirty-Second AAAI Conference on Artificial Intelligence*.

[40] Jiacheng Xu, Kan Chen, Xipeng Qiu, and Xuanjing Huang. 2016. Knowledge graph representation with jointly structural and textual encoding. *arXiv preprint arXiv:1611.08661* (2016).

[41] Bishan Yang, Wen-tau Yih, Xiaodong He, Jianfeng Gao, and Li Deng. 2014. Embedding entities and relations for learning and inference in knowledge bases. *arXiv preprint arXiv:1412.6575* (2014).

[42] Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, et al. 2020. Big Bird: Transformers for Longer Sequences.. In *NeurIPS*.

[43] Guineng Zheng, Subhabrata Mukherjee, Xin Luna Dong, and Feifei Li. 2018. Opentag: Open attribute value extraction from product profiles. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 1049–1058.

[44] Huaping Zhong, Jianwen Zhang, Zhen Wang, Hai Wan, and Zheng Chen. 2015. Aligning knowledge and text embeddings by entity descriptions. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*. 267–272.

CHEX: Multiversion Replay with Ordered Checkpoints

Naga Nithin Manne*
Argonne National Lab.
Lemont, IL, USA
nithinmanne@gmail.com

Shilvi Satpati
DePaul University
Chicago, IL, USA
ssatpati@depaul.edu

Tanu Malik
DePaul University
Chicago, IL, USA
tanu.malik@depaul.edu

Amitabha Bagchi
IIT, Delhi
Delhi, India
bagchi@cse.iitd.ac.in

Ashish Gehani
SRI
Menlo Park, CA, USA
ashish.gehani@sri.com

Amitabh Chaudhary
The University of Chicago
Chicago, IL, USA
amitabh@uchicago.edu

ABSTRACT

In scientific computing and data science disciplines, it is often necessary to share application workflows and repeat results. Current tools containerize application workflows, and share the resulting container for repeating results. These tools, due to containerization, do improve sharing of results. However, they do not improve the efficiency of replay. In this paper, we present the multiversion replay problem, which arises when multiple versions of an application are containerized, and each version must be replayed to repeat results. To avoid executing each version separately, we develop **CHEX**, which checkpoints program state and determines when it is permissible to reuse program state across versions. It does so using system call-based execution lineage. Our capability to identify common computations across versions enables us to consider optimizing replay using an in-memory cache, based on a checkpoint-restore-switch system. We show the multiversion replay problem is NP-hard, and propose efficient heuristics for it. **CHEX** reduces overall replay time by sharing common computations but avoids storing a large number of checkpoints. We demonstrate that **CHEX** maintains lightweight package sharing, and improves the total time of multiversion replay by 50% on average.

PVLDB Reference Format:
Naga Nithin Manne, Shilvi Satpati, Tanu Malik, Amitabha Bagchi, Ashish Gehani, and Amitabh Chaudhary. CHEX: Multiversion Replay with Ordered Checkpoints. PVLDB, 15(6): 1297-1310, 2022.
doi:10.14778/3514061.3514075

PVLDB Artifact Availability:
The source code, data, and/or other artifacts are available at https://github.com/depaul-dice/CHEX.

1 INTRODUCTION

Suppose that Alice is researching different image classification pipelines. She has a large labeled set of images and progressively tries different combinations of preprocessing steps and neural network architectures. For example, she may replace an entire step with one that is more sophisticated but slower, or vice-versa. As she makes changes, she keeps a copy of the previous versions in separate Jupyter notebooks. We call these her different *program versions; they are similar to different* experiments in scientific computing. Once done, Alice would like to share her different program versions with Bob so that he can independently repeat and regenerate the results and verify Alice's work. We say Bob faces the *multiversion replay problem*: executing all the versions given to him by Alice as efficiently as possible where (i) many versions repeat some of the same preprocessing steps, but (ii) without reusing any of Alice's own computation. In this paper, we address this problem.

Collaborative scenarios such as the one above arise routinely in scientific computing and data science, where sharing, repeating, and verifying results is common. Several tools have been recently proposed for sharing and reproducing such scenarios [1, 8, 22, 41–44, 57]. These tools audit the execution of a program, and create a container-like package consisting of all files referenced by the program during its execution. This package can then be used to repeat results in different environments. These tools have much to offer; they do not, however, exploit the efficiency possible by solving the multiversion replay problem. As the reproduction of results becomes increasingly time consuming [53], addressing such problems is critical.

One of the above tools is the Sciunit system [1, 55], developed by some of the co-authors. It allows multiple versions of a program to be included in the same package and shared for repetition. We noted that having two or more versions in the same package sets up a natural opportunity for reusing computations that are often common across versions—*i.e.,* a number of versions may perform the same computations for quite some time before they branch out as the researcher tries out different options. But, in order to accurately identify computations that can be reused across versions, we need to be able to determine the point to which the execution of two versions can be treated as equivalent and from which point the execution branches. We develop a methodology to identify common computations in program code fragments or *cells* of the versions. This methodology depends on lineage audited during program execution [38, 42, 43]. For repetition, at Bob's end, we share computational state across versions in the form of checkpoints. Let us now see with an example how sharing computational state across versions via checkpoints creates an opportunity to optimize computational time when repeating multiple versions.

Suppose that Alice has shared with Bob a package with three versions. Assume Alice has developed her code using a notebook,

*Work done as part of a summer internship at DePaul.

Proceedings of the VLDB Endowment, Vol. 15, No. 6 ISSN 2150-8097.
doi:10.14778/3514061.3514075

and her first version is divided into two cells that take time 1 minute and 10 minutes, respectively (Figure 1 left). Her second version has the same first two cells but she adds a third cell. Her last version, which processes the dataset, has the same first cell, but diverges after that, and cells 2 and 3 now takes 11 and 2 minutes, respectively. Now, during repetition, if Bob checkpoints cell b while computing $v1$ and then restores that checkpoint for $v2$, he can complete $v2$ in just 1 unit of time (saving 11 units in $v2$). This checkpoint is, however, not useful for $v3$ since cell b has been changed to d in this version. Observe, that although a is common across all three versions, checkpointing a, instead of b, is not optimal. In the example on the right, on the other hand, checkpointing a is the better option since the bulk of the computation takes place in a. The example

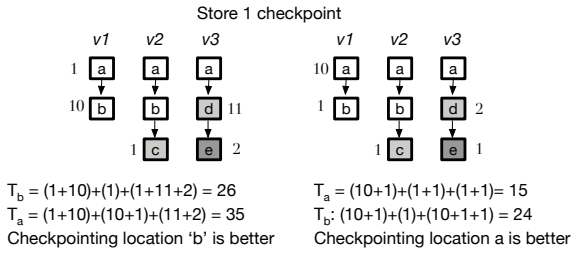


Figure 1: Deciding checkpoint location depends on cost and size estimates of the cells.

above leads to the question: *Why doesn't Bob just checkpoint both a and b?* Storage is cheap after all! Indiscriminate caching, however, is not a practical solution: In machine learning examples, e.g., checkpointing all cells across versions (as our experiments indicate) can lead to a memory requirement in the range of 50-550GB, for even moderately-sized multiversion programs. Thus, we consider a limited in-memory space as this avoids additional I/O costs from checkpointing. Limited space leads to an optimization problem: As we will show, given limited space and multiple versions with different cost and size estimates of cells, deciding checkpoint locations for efficient replay of all versions is an NP-hard problem. We present efficient heuristics to solve this problem.

The **CHEX** *system.* We present **CHEX**, a system for efficient multiversion replay that uses recorded lineage shared in container-like auditing systems to (i) determine when the program state is identical across versions, and (ii) decides which common computations to save in an in-memory, limited-size cache, and where to continue recomputing. Effectively, **CHEX** computes an efficient plan for Bob to use his cache to repeat Alice's multiversion program with minimum computation cost. Subsequently **CHEX** repeats the computation according to this plan.

To execute the multiversion program, we first need a plan for sharing and reusing computational state across versions. A possible approach would be to reuse program elements such as the output of functions, expressions, or jobs. Such reuse approaches were examined in [16–18]. However, these methods make assumptions about the programs—they are limited to programs with no side-effects and apply to specific (functional or interpreted) programming languages. Such assumptions are too restrictive in a sharing scenario.

Our approach is program-agnostic and we, instead, use checkpoints. A checkpoint saves the computational state at a specific program location so that the *same* program can be restored from the location at a later time. To share computations, we extend *checkpoint-restore* to *checkpoint-restore-switch*, in which a system checkpoints a common computational state and and restores it later to resume a *different* version of the program.

The challenge of checkpoint-restore-switch, however, is determining locations at which to checkpoint, since ideally programs may be checkpointed after each instruction. Even if we decide at a fixed number of program locations, before reusing a checkpoint we must verify that two versions share the same computational state at a given program location. *In this paper we solve this dual challenge by showing that when a program is divided into* cells*, computational state can be shared across versions by using fine-grained execution lineage.* Dividing a program into cells is used in read-evaluate-print (REPL) programming environments, which are increasingly popular [26]. However, **CHEX** does *not* necessitate that a user employ REPL style; it transparently divides a program into REPL-style cells. This paper contributes the following:

Maintains lightweight package sharing. **CHEX** does not require users like Alice to share checkpoints as part of the shared package. Instead, **CHEX** audits the execution of each version to record execution details. We note that in reproducibility settings, it is not desirable to allow Alice to share her checkpoints since that defeats the purpose of reproducibility.

Merging versions based on lineage. **CHEX** compares fine-grained lineage to check if the program state is common across cell versions. It combines versions into an *execution tree*.

Deciding checkpoint location. Given that the multiversion replay problem under space constraints is computationally intractable, we rely on depth-first-search (DFS) traversals of the execution tree to help us identify a subset of possible checkpointing decisions for the execution units of the program. We call the members of this subset *DFS-based replay sequences.* We propose two heuristic algorithms for deciding which cell state to checkpoint such that the multiple versions can be replayed in a minimum amount of time.

Experiments on real and synthetic datasets. We experimented with real machine learning and scientific computing notebooks as well as synthetic datasets, showing that **CHEX** improves the total time of multiversion replay by 50%, or correspondingly replays twice the number of versions in a given amount of time. We show that the overheads of creating execution trees is significantly lower than the gain from replay efficiency.

Working prototype system: We have developed a prototype **CHEX** system, which given an execution tree performs multiversion replay. **CHEX** currently uses standard auditing methods, developed by us [1, 55], to build execution trees and determine cell reuse. For multiversion replay, we extended these methods to work with interactive Jupyter notebooks, as well as, transform regular programs to REPL-style computation via code-style paragraphs.

2 BACKGROUND

We briefly describe the REPL environment under which **CHEX** operates. **CHEX** is not limited to the REPL environment but this is easy to illustrate visually so we adopt this for ease of exposition. We discuss generalization to other environments in Section 9.

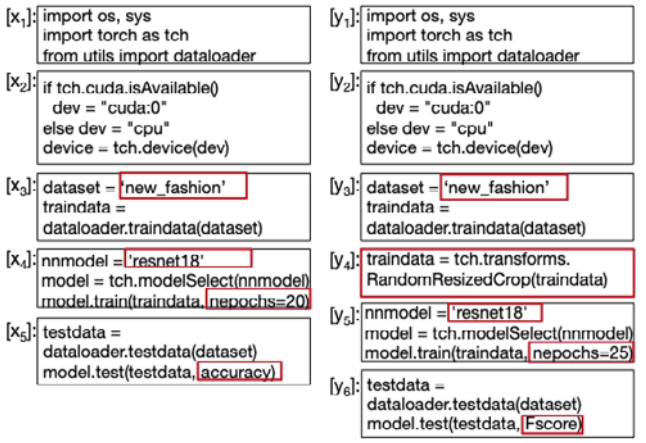


Figure 2: An illustration of REPL programs. The left program L_1 trains a machine learning model (resnet18) on a training dataset and evaluates its accuracy on a test dataset. The right program L_2 is the same, except that it adds a preprocessing step to the training dataset. L_1 has 5 cells, L_2 has 6 cells.

REPL or Read-Evaluate-Print-Loop is a programming environment. A popular example is the Jupyter notebook. As shown in Figure 2, it contains code partitioned into *cells*. Developers typically use a separate cell for each "step" of the program: preprocessing the dataset, training the model, etc. This allows them to interactively test each step before writing the next. One restriction is that control flow constructs, such as if-blocks, loops, cannot be split across cells. We denote a REPL program by an ordered list of cells, *e.g.*, the left program in Figure 2 is denoted $L_1 = [x_1, x_2, \ldots, x_5]$, and the right program as $L_2 = [y_1, y_2, \ldots, y_6]$.

In a typical REPL execution, cells are executed in sequence from the first to last. While the Jupyter notebook allows out-of-order cell execution, we do not consider such execution. (We elaborate on this constraint further in Section 9.) The state of the program at the end or beginning of each cell is termed the program state. The program state at any point of execution consists of the values of all variables and objects used by the program at that point — intuitively, it is all the contents of the memory associated with the program. So, e.g., for the program $L = [x_1, x_2, \ldots, x_5]$, the corresponding program states are $[ps_0, ps_1, \ldots, ps_5]$, in which ps_{i-1} denotes the program state just before cell x_i is executed. The state ps_0, which is just before the first cell is executed, includes the value of the environment and any initial input.

CHEX works in combination with an *auditing system* which monitors executions and provides the following details about each program state, ps_i:

- **computation time**, δ_i, the time to reach the program state ps_i from its predecessor ps_{i-1},
- **size**, sz_i, size of the program state ps_i,
- **code hash**, h_i, computed by hashing code in cell i, and
- **lineage**, g_i, which is determined by combining the predecessor cell's lineage with the sequence of system events that are triggered by program instructions in the cell i and the hashes of the associated external data dependencies. Thus, $g_i = (g_{i-1}, h_i, E_i)$, where E_i is the ordered set of system events in cell along with the hash of the content accessed by the event. Initially, $g_0 = \{\}$.

To see why g_i is defined so, we note that the execution of the program code in cell i (and the code in previous cells) resulted in ps_i. Therefore, ps_i at the end of a cell's execution depends on its (i) initial environment, (ii) code that is run, and (iii) external input data. The environment is determined by the execution state at the start of the cell. Thus, (i) and (ii) are captured via g_{i-1} and h_i. Further, every external input data file f is accessed via a system call event. For each such event, we record a hash of its contents of f in E_i.

Figure 3 shows the audited information for the two programs, L_1 and L_2. The ordered set of system events for the third cells of the two programs are shown in the shaded box below.

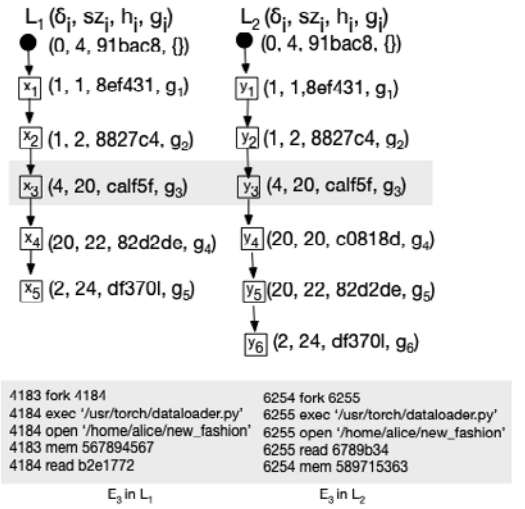


Figure 3: Auditing of programs L_1 and L_2 in terms of δ, sz, h, g. Events in E_3 show a forked process, open of an external file along, and read of data content, denoted by its hash value. We determine how to check if E_3 across versions is equal in Section 6.

3 CHEX OVERVIEW

As we see in Figure 2, the two programs behave the same till the end of the third cell (x_3 in L_1, y_3 in L_2) and then diverge. If the audited lineage, as shown in Figure 3, is established to be the same, then the program state at the end of x_3 can be used before y_4. i.e, we can skip executing cells y_1 to y_3. **CHEX** uses recorded lineage to determine when the program state is identical across versions, and decides which common computations to save. We now present a high-level block diagram of **CHEX** in Figure 4.

CHEX has two modes: audit and replay. It is used in audit mode to audit details of executions on Alice's side. Details of multiple executions, i.e. the δ, sz, h and g of each cell across versions are represented in the form of a data structure called the *Execution Tree*. We discuss the execution tree and how it is created in detail in Section 6. **CHEX** creates a package of all Alice's versions and their data, binary, and code dependencies, along with the execution tree. This package can now be shared with Bob.

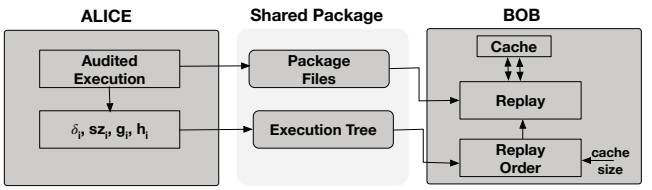


Figure 4: CHEX Overview.

CHEX is used in replay mode on Bob's side. It first determines an efficient *replay sequence* or replay order, i.e., a plan for execution that includes checkpoint caching decisions. To do so **CHEX** inputs a cache size bound, B, and then executes a heuristic algorithm on the execution tree received from Alice to determine the most cost efficient replay sequence for that cache size. Computing a cost-optimal replay sequence for multiple versions of a program with a cache bound is an NP-hard problem as we show in Section 4 and so we describe some efficient heuristics for this purpose (Section 5). Finally, once the replay sequence is computed, **CHEX** uses this replay sequence to *compute, checkpoint, restore-switch* REPL program cells or *evict* stored checkpoints from cache.

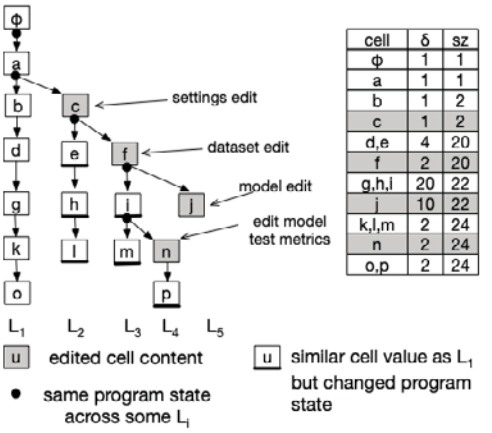
Our assumptions. Our basic assumption is that Bob wishes to independently verify the results from Alice's versions but is time constrained to repeat all her versions. We do not make any assumptions on the types of edits that differentiates one version from the next. Thus, Alice can change values of parameters, specifications of datasets, models, or learning algorithms. She can also add or delete entire cells. We illustrate possible changes via red boxes (Figure 2) across program versions. We only assume that edits result in valid executions, which do not terminate in an error, and, each version is executed in the natural order, top to bottom.

4 THE MULTIVERSION REPLAY PROBLEM

We now describe the multiversion replay problem. Figure 6 summarizes the symbols used in Section 2. In the replay mode, **CHEX** inputs an execution tree, T, and a fixed cache size, B, to solve the multiversion replay problem. We define the execution tree as:

Definition 1. *(Execution Tree) An execution tree $T = (V, E)$ is a tree in which each program state is mapped to a node and equal program states across the different versions are mapped to the same node. Each root to leaf path in T corresponds to a distinct version L_i.*

Example. Figure 5 shows the execution tree created from five versions. In this tree, each root to leaf path corresponds to version L_i. In L_1 there is an edit to settings of the program at cell b, resulting in L_2 and a branch at a, the last common node across L_1 and L_2. Similarly, in L_3 there is a dataset change to L_2 at cell e, resulting in L_3 and a branch at c, the last common node across L_2 and L_3. The common nodes till a branch in the tree correspond to the subsequence of cells that are equal across versions. The tree branches at a cell node, subsequent to which cells are not reusable. **CHEX** computes cell equality using execution lineages. We will discuss how this is done via system calls in detail in Section 6. Intuitively, establishing cell equality makes program states reusable across versions.



cell	δ	sz
Φ	1	1
a	1	1
b	1	2
c	1	2
d,e	4	20
f	2	20
g,h,i	20	22
j	10	22
k,l,m	2	24
n	2	24
o,p	2	24

Figure 5: The enhanced specifications of versions (without lineages for simplicity) represented as a tree. The cell i appears similar to h but has a changed program state due to edited f. Both m and n proceed from i's state.

The multiversion replay problem is an optimization problem that arises when multiple versions of a program, each previously executed, are replayed as a collection. Once the multiversion program is represented as an execution tree it is clear that there is some advantage in not replaying the common prefixes of this tree.

Example. If we replay the five versions of Figure 5 sequentially we incur total cost of 129. On the other hand, assuming a cache size of 25, if we store the *checkpoint* at common prefixes, *restore-switch* the checkpoint later to avoid computing the common prefix for the next version, and *evict* the previous checkpoint to store a new one, the replay cost is reduced to 114 as shown in the first replay sequence of Figure 7. In the second figure we see that a different set of checkpointing decisions can improve the cost even when the cache size remains the same. Finally we see that increasing the space to 50 further improves replay costs to 95.

Under these operations, and given an execution tree and a fixed amount of space for storing checkpoints, the multiversion replay problem aims to determine a replay sequence that has the minimum replay costs. We define a general replay sequence as follows:

Definition 2 (Replay sequence). *Given execution tree $T = (V, E)$ and a cache of size B, a replay sequence R consists of m steps such that step t specifies the operation O_t performed and the resulting state of the checkpoint cache S_t, i.e.,*

$$R = [(O_t, S_t) : 0 \leq t \leq m]$$

We will use the term replay order interchangeably with the term replay sequence.

At the initial step S_0 is empty and the root of the tree is computed. At any given step t, O_t is of one of the following four types. Here, u_j and u_k are nodes in V, the vertices of T.

- Compute $CT(u_j)$: computes u_j;
- Checkpoint $CP(u_j)$: checkpoints u_j into the cache;
- Restore $RS(u_j, u_k)$: restores a previous checkpointed u_j in cache and switches to u_k where $u_j = parent(u_k)$; and
- Evict $EV(u_j)$: evicts a previous checkpoint u_j from cache;

L_j	REPL program version
x_i	Cell in L_j
h_i	Hash of source code in x_i
g_i	Cumulative hash of source code and ext. dependencies till x_i
ps_i	Program state at end of x_i
sz_i	Size of ps_i
δ_i	Computation time to reach ps_i from ps_{i-1}
T	Execution tree combining overlapping L_j's
B	Fixed cache size
R	Replay sequence for T
O_t	Operation at step t in R
S_t	Set of program states in cache after step t in R
$\delta(R)$	Computation time for R

Figure 6: Notation used in Section 2 and Section 4

The cache size can never exceed B, i.e. $|S_t| \leq B$ for $0 \leq t \leq m$. Further, an operation O_t at step t can only be performed on u_j, u_k under the following constraints:

- *Checkpoint from working memory:* A node in the execution tree is checkpointed only if it was computed in some previous step, after which there are only some evictions (to make space), if at all, *i.e.,* if $O_t = CP(u_j) \implies S_t = S_{t-1} \cup \{u_j\}$ and $O_{t-i} = CT(u_j)$, for some $1 \leq i \leq t$, and $O_{t'} = EV(u_{t'})$ for $t - i < t' < t$.
- *Restore from cache and switch to child:* A node is restored only if it was in cache in a previous step, and without altering cache state, switches to one of its children in the execution tree, which is computed next *i.e.,* if $O_t = RS(u_j, u_k) \implies u_j \in S_{t-1}, S_t = S_{t-1}, O_{t+1} = CT(u_k)$.
- *Evict from cache:* A node is evicted from cache and alters its state, *i.e.,* if $O_t = EV(u_j) \implies u_j \in S_{t-1}, S_t = S_{t-1} - \{u_j\}$.
- *Continue computation:* Continue computing a node if its parent was being computed or if its parent was restored, *i.e.,* if $O_t = CT(u_j) \implies O_{t-1} = CT(parent(u_j))$ or $O_{t-1} = RS(parent(u_j), u_j)$ and $S_t = S_{t-1}$ or t= 1 and u_j=root of the tree T.

We assume the operations generate *complete* and *minimal* sequences. A replay sequence is complete if all leaf nodes of the tree T appear in R, and is minimal if no u_j that is in cache is recomputed.

Problem 1 (The Multiversion Replay Problem (MVR-P)). *Given tree $T(V, E)$, the multiversion replay problem is to find a complete replay sequence R that minimizes*

$$\delta(R) = \sum_{i=0}^{|R|=m} \delta_{O_t},$$

in which $\delta_{O_t} = \delta_j$, when $O_t = CT(u_j)$, and $\delta_{O_t} = 0$ otherwise.

In MVR-P, we assume the cost of checkpoint, restore-switch, and evict operations to be negligible. Thus, the only cost considered is the cost of computing the cells. Determining the minimum cost replay order leads to a natural trade-off between computational cost of cells and fixed-size cache storage occupied by the checkpointed state of the cells. Thus, to optimally utilize a given amount of storage we must determine for each cell whether its next cell be recomputed, or some other cell be recomputed by checkpointing the state of the current cell. We state that determining the replay order is computationally hard.

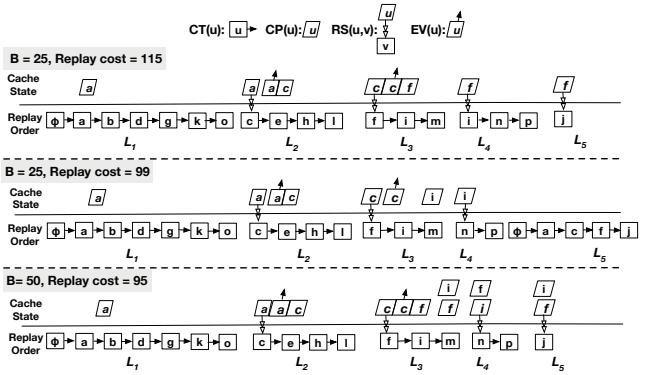


Figure 7: Replay sequences for the execution tree in Figure 5 showing use of operations and the state of the cache.

Theorem 1. *MVR-P is NP-hard.*

We show that the decision version of MVR-Pis NP-hard. Given an execution tree T, a cache size parameter $B > 0$ and a total cost parameter $\Delta > 0$, define $RP(T, B, \Delta)$ to be the decision problem with answer YES if there is a replay sequence of T with cost at most Δ and size of cache at most B, and with answer NO otherwise.

The proof is by reduction from the decision version of bin packing. In outline, the proof works by constructing an execution tree whose depth 1 nodes have checkpoint sizes corresponding to the size of the items to be packed into bins in the bin packing problem. The B of $RP(T, B, \Delta)$ is set to the size of the bins. In order to force caching, we keep Δ small and add nodes below the depth 1 nodes so that each of the level one nodes has to be cached when first computed. We are able to show that by carefully adding subtrees below the depth 1 nodes we are able to prove a tight relationship between the two problems, i.e., the bin packing decision problem gives a Yes answer iff $RP(T, B, \Delta)$ gives a yes answer.

We omit the proof due to space restrictions, referring the reader to the full version of this paper available at [34].

5 HEURISTIC SOLUTIONS

From Theorem 1 we know that it is unlikely we will find polynomial time solution to MVR-P. Accordingly, we present two efficient heuristics for this problem. Both heuristics restrict our exploration of the search space to solutions in which the execution order of the nodes of the execution tree corresponds to a DFS traversal of the tree—a natural, simple order in which to approach the replay of the tree. In order to formalize this notion we present some definitions. In the following, for sake of brevity, we specify only the compute $CT(u_j)$ type operations in replay sequences. The other operations (checkpoint, restore, evict) are separately specified. In this briefer format, each step of a replay sequence is of the form (u_t, S_t) specifying that at step t, u_t is computed, and the resulting cache is S_t.

Definition 3 (Ex-Ancestor replay sequence). *Suppose $T = (V, E)$ is an execution tree. Given any replay sequence $R = \{(u_t, S_t) : 1 \leq t \leq T\}$ we define its* first appearance order *to be $i_1 < i_2 < \cdots < i_{|V|}$ such that u_{i_j} is the first appearance of a node v_j of T in R. We call the indices $i_1, \ldots, i_{|V|}$ as* first appearances *and all other indices as*

repeat appearances. *For a replay sequence R, for each $j \in \{2, \ldots, |V|\}$ the sequence of cells $u_{i_{j-1}+1} \ldots u_{i_j-1}$ is called the* helper sequence *for v_j. If the helper sequence for v_j forms a path from an ancestor of v_j to v_j for each j then R is called an* ex-ancestor replay sequence.

We observe that in an ex-ancestor replay sequence if the helper sequence of v_j is non-empty then it either begins with the root of T or with a node whose parent is in $S_{i_{j-1}}$.

We illustrate this definition with an example. Consider the tree in Figure 5. Assume for now that cache size $B = 0$ and consider the following replay sequence:

$$a, b, d, g, k, o, \mathbf{a}, c, e, h, l, \mathbf{a}, \mathbf{c}, f, i, m, \mathbf{a}, \mathbf{c}, \mathbf{f}, \mathbf{i}, n, p, \mathbf{a}, \mathbf{c}, \mathbf{f}, j$$

where bold font indicates repeat appearance nodes. Here the indices 1, 2, 3, 4, 5, 6, 8, 9, 10, 11, 14, 15, 16, 21, 22, and 26 are first appearances and all others are repeat appearances. Let's take the example of node n. It's first appearance index is 21. Its helper sequence extends from indices 17 to 20 and contains $\mathbf{a}, \mathbf{c}, \mathbf{f}, \mathbf{i}$. This is a simple path in the tree beginning from the node containing a which is an ancestor of n. In fact it is easy to verify that this sequence is an ex-ancestor replay sequence.

The question arises: Are there meaningful replay sequences that are not ex-ancestor replay sequences? For example, would it make sense to modify n's helper sequence and make it $\mathbf{a}, \mathbf{c}, e, \mathbf{f}, \mathbf{i}$. It appears that the extra computation of e is superfluous and so a priori it is not obvious that such replay sequences are meaningful from the point of view of efficient replay. Therefore we focus on ex-ancestor replay sequences. We conjecture that an optimal solution to MVR-P will be such a sequence.

Definition 4 (DFS-based replay sequence). *Suppose $T = (V, E)$ is an execution tree for a collection of traces C. We say a complete and minimal replay sequence R is a* DFS-based replay sequence *if R is an ex-ancestor sequence and the first appearance order of R is a DFS-traversal order of T.*

Note that first appearance sequence of the example discussed below Definition 3 gives us a DFS-traversal of T. Hence this is a DFS-based replay sequence for the tree of Figure 5.

Now assume a cache size $B = 25$ and the caching decisions made according to the first replay sequence in Figure 7.[1] The corresponding replay sequence is similarly: $a, b, d, g, k, o, c, e, h, l, f, i, m, \mathbf{i}, n, p, j$. Since a, c, and f are cached at appropriate junctures, the only node with a non-empty helper sequence is n and the length of this sequence is just one, i.e., there is only one cell that has to be recomputed apart from its first appearance computation. For the second and third sequence in Figure 7 the number of recomputations are similarly three $(\mathbf{a}, \mathbf{c}, \mathbf{f})$ and zero respectively.

We are able to explicitly bound the number of DFS-based replay sequences.

Proposition 1. *Suppose $T = (V, E)$ is an execution tree for a collection of traces C such that $|V| = n$ and the height of T is h. Let b_u be the number of children of node $u \in V$ and let*

$$\bar{b} := \frac{1}{n} \sum_{u \in V} b_u \log b_u.$$

Then, the number of DFS-based replay sequences of T is $O(2^{n(h+\log h+\bar{b})})$.

[1] All three replay sequences in Figure 7 are DFS-based.

Proof. Let us fix a DFS traversal order. The helper sequence preceding (and including) each node can be at most h in length and hence the length of a replay sequence can be no longer than hn. Since each helper sequence is an ex ancestor path to a node of the tree, we can have at most h choices of a helper sequence at each node. Therefore there are at most h^n different sequences that can qualify to be DFS-based replay sequences. Note that at each point of one of these sequences of cells we can decide to either cache the cell that we have just computed (this may require the eviction of something previously cached) or to not cache it. Hence there are at most 2^{nh} replay sequences associated with each of the h^n different sequences that we got for a single DFS traversal order. This gives us an upper bound.

To compute the number of DFS traversal we simply permute the children visited by DFS at each step to get $\prod_{u \in V} b_u!$ which can be rewritten as $2^{n\bar{b}}$ by using Stirling's approximation. Multiplying we get the result. □

Since h is $\Omega(\log n)$ from Proposition 1 it appears that the space of possible solution is superexponential. In order to control the complexity of our solutions we restrict the solution space in two different ways and define two heuristics.

5.1 Persistent Root Policy Greedy Algorithm

Within the space of DFS-based replay sequences, for our first heuristic, we propose the following caching policy: *A cell can be cached only when it is first computed. Once cached the cell remains in the cache till every leaf of the subtree rooted at the node containing cell is computed.* We call this the *DFS Persistent Root policy.*

Given a DFS traversal order this policy reduces the size of the solution space to $O(2^n)$ which is still exponential in the size of the tree. We present a greedy algorithm called *Persistent Root Policy Greedy* (**PRP**) that helps find a good solution in polynomial time. We present the listing of **PRP** as Alg. 1. The algorithm begins with the baseline cost (stored in min) of a DFS-based replay sequence in which no node is cached and seeks out the node of the tree whose addition to the list S achieves the maximum improvement over the baseline. This process continues incrementally while it is possible to include another node in the list. The process will stop when the subroutine *DFSCost* tells us that there is no node remaining in $V \setminus S$ that can be included in S. Typically this will happen because for every node u remaining in $V \setminus S$, the cache will be full when it is encountered in the DFS order. In such a situation *DFSCost* will return ∞. This algorithm takes $\theta(n^2)$ time to find each candidate to include in the list of nodes to be cached, and there are potentially $O(n)$ such nodes. Therefore the time complexity of this algorithm is $O(n^3)$. However there is no guarantee of optimality.

PRP is a greedy algorithm that seeks, at each iteration, to pick for caching the vertex of the execution tree that minimizes the cost. However, it can be easily modified to choose a vertex that minimizes the cost incurred *per unit of cache memory consumed.* Normalizing by size is a common measure for object caches [7, 32]. We experimentally study both these variants in Section 7. We will refer to the cost-minimizing version as **PRP-v1** and the ratio minimizing version as **PRP-v2**.

Algorithm 1 A greedy algorithm that takes as input execution tree T, and a cache size parameter B. It outputs list S of nodes to be cached under the DFS Persistent Root policy.

```
1:  function PRP(T, B)
2:      S ← ∅
3:      f ← True ▷ f is True while greedy is able to extend its solution
4:      r ← root(T)
5:      Set min ← DFSCost(r, S, B, 0) ▷ The function gives us the cost of a DFS-
        based replay sequence for T given a list of nodes S that must be cached when first computed.
6:      while f is True and S ≠ V do
7:          f ← False
8:          for each u ∈ V \ S do
9:              if DFSCost(r, S ∪ {u}, B, 0) < min then
10:                 f ← True ▷ We can extend the solution
11:                 u* ← u ▷ u* is the current best candidate
12:         if f is True then
13:             S ← S ∪ {u*}
14:             min ← DFSCost(r, S, B, 0)
15:     return S
1:  function DFSCost(u, S, B, b) ▷ u is a node of T; b is the cache budget used by
    the path from the root of T to u. Called with u = root(T) and b = 0 this returns the cost of
    computing the entire tree.
2:      if u ∈ S and b + sz_u > B then
3:          return ∞ ▷ Cache size infeasibility detected
4:      c_u ← cost of computing u from nearest ancestor in S
5:      if u has no children then
6:          return c_u
7:      sum ← 0
8:      for each v that is a child of u do
9:          if u ∈ S then
10:             sum ← DFSCost(v, S, B, b + sz_u)
11:         else
12:             sum ← DFSCost(v, S, B, b) + c_u ▷ u is not cached so must be
        recomputed for each child
13:     if u ∈ S then
14:         sum ← sum + c_u ▷ u must be computed once
15:     return sum
```

5.2 Parent Choice Algorithm

We now present a second heuristic that, while still not being optimal, searches a superset of the portion of the solution space searched by **PRP**. For each $u \in V$ it seeks to partition the children of u into two sets: P_u of nodes for which it is better to cache u for the computation of the corresponding child subtrees, and $\overline{P}_u$ for which it is not. As in Persistent Greedy, caching choices once made persist here as well.

The listing of the essential recursive Parent Choice is presented as Alg. 2. When called with (u, S) we explore the situation in which we are given the set S of ancestors of u that will be in cache while the subtree rooted at u is computed. In case u happens to be a leaf, no further decisions are needed, and we simply return the cost of computing u given cache S (Lines 2-4). Else, we need to determine what is best for each child u_i of u: Should the subtree rooted at u_i be computed with S as is, or is it better to augment the cache with u (denoted S_{+u}). In the former the subtree may be forced to recompute u multiple times, in the latter cache space which may be more useful down the subtree is used up. The two costs are computed recursively (Lines 14-15), and the child is assigned to the set P_u or $\overline{P}_u$ corresponding to the lower cost (Lines 16-19)). Note that when adding u to the cache is infeasible, i.e. $|S_{+u}| > B$, we make the first choice for each node, i.e. assign them all to P_u. (Lines 7-10). Finally, we return the cost value up the recursion stack (Line 20).[2]

Algorithm 2 A recursive algorithm the computes for a tree rooted at u the lowest DFS-based replay cost for a given cache S. The child subtrees of u are allowed to choose between executing with S or in addition caching u.

```
1:  function ParentChoice(u, S)
2:      if u is a leaf then
3:          a ← nearest ancestor of u in S
4:          return cost of computing u from a
5:          ▷ If a doesn't exist, return cost of computing u from scratch.
6:      S_{+u} ← S ∪ {u} ▷ S_{+u} is cache that also includes u.
7:      if size of cache S_{+u} > B then
8:          ▷ Caching u is not a option; process its children with S.
9:          P_u ← ∅; P̄_u ← Children(u).
10:         return Σ_{u_i ∈ P̄_u} ParentChoice(u_i, S)
11:     P_u ← ∅, P̄_u ← ∅
12:     ▷ P_u will collect the nodes for which caching parent u is cheaper.
13:     for each u_i ∈ Children(u) do
14:         cost(u_i, S_{+u}) ← ParentChoice(u_i, S_{+u})
15:         cost(u_i, S) ← ParentChoice(u_i, S)
16:         if cost(u_i, S_{+u}) ≤ cost(u_i, S) then
17:             P_u ← P_u ∪ {u_i}
18:         else
19:             P̄_u ← P̄_u ∪ {u_i}
20:     return Σ_{u_i ∈ P_u} cost(u_i, S_{+u}) + Σ_{u_i ∈ P̄_u} cost(u_i, S)
```

The essential recursive algorithm **PC** needs to be implemented using standard dynamic programming memoization and backpointers (see, e.g., [11]). Once a call with input (u, S) is complete, the corresponding return cost value and the two sets P_u and $\overline{P}_u$ are recorded. The initial call is with $(\text{root}(T), \emptyset)$. This returns the cost of the optimal replay sequence for the entire T. To construct the replay sequence itself, "follow the backpointers": Start with $u = \text{root}(T)$ and $S = \emptyset$. If for the corresponding call, P_u is not empty, compute u (possibly by restoring the closest ancestor in the current cache) and checkpoint it. Update S to include u. Then recursively compute the subtrees rooted at the nodes in P_u. Next update S to remove u. Following this, recursively compute the subtrees rooted at the nodes in $\overline{P}_u$, if any.

The above implementation takes time and space proportional to the total number of child nodes encountered over all recursive calls. For each $u \in V$, at most one recursive call is made for each possible set of ancestors in the cache. The number of different ancestor sets is at most 2^h. Thus the total time taken is $O(2^h \sum_{u \in V} b_u)$.

[2] We do not explicitly show the cost of computing u in order to cache it for P_u. This cost is offset by the same cost incurred by the first child subtree in $\overline{P}_u$, as shown, but not actually paid since u is already in cache when it is executed. The case of $\overline{P}_u = \emptyset$ has a further optimization that is possible; see the full version of this paper [34].

6 EXECUTION TREE

We now discuss how **CHEX** constructs the execution tree at Alice's end. As per Definition 1, an execution tree merges equal program states of different versions into a single node in the tree. Given the per cell values of state computation time δ_i and size sz_i, state lineage g_i, and state code hash h_i, we use the following conditions to identify equal program states:

Definition 5 (State equality). *Given two program versions L_1 and L_2, state ps_i in L_1 is equal to state ps_j in L_2, denoted $ps_i = ps_j$, if and only if (i) $h_i = h_j$, (ii) $g_i = g_j$, and (iii) δ and sz costs are similar.*

In other words we say that two states are equal if they are reusable *i.e.,* they are (i) equal at code syntactic level, (ii) after cell execution, result in the same state lineage (note state lineage of i^{th} cell depends on state lineage of previous cell), and (iii) have roughly similar execution costs. Program state does not remain equal when cell code is edited, which changes the hash value of that cell and any subsequent cell. Similar states across versions also do not remain equal if costs change drastically, *i.e.*, computed on different hardwares (*viz.* GPU vs CPU). Equating state lineage depends on the granularity at which the system events are audited. Since in **CHEX**, lineage is audited at the level of system calls, there are some pre-processing steps that are necessary to establish equality, such as accounting for partial orders, abstracting real process identifiers, and accounting for hardware interrupts. We describe these issues below.

Lineage equality implies that at end of cell i of version L_1, g_i is the same as that at end of cell i of version L_2. This is true if and only if the *sequence* of system call events (and their parameters)—till i in L_1 and i in L_2—exactly match. But if a cell, e.g., forks a child process, which itself issues system calls, then each version's sequence will contain the parent calls and the child process calls interleaved in possibly different orders.

In Figure 3 the parent process forks a child and then issues a 'mem' memory call, and the child process itself issues 'exec', 'open', and 'read' calls. As the figure shows, it is possible that in the sequence for version L_1 the 'mem' access is before the 'read', while for L_2 it is after. If we want to correctly determine that the state in L_1 is identical to that in L_2 at this point, we need to recognize that the sequence of system calls is an arbitrary total order imposed on an underlying partial order. The partial order for L_1 and L_2 is identical, while the total order can differ.

In our implementation, we reconstruct the underlying partial order when we detect asynchronous computation, and match it to identify equality of program states in different versions. This is achieved by separating the events into PID-specific sequences and then comparing corresponding sequences. The above comparison is established by abstracting process identifiers to their logical values. Memory accesses cannot be abstracted and we just count the number of accesses in a cell. Comparison must also account for external inputs in addition to system events. As Figure 3 shows the hash of external dataset file 'new_fashion' is changed from 'b2e1772' to '6789b34. Thus, the two cells cannot be equated even though the order of system call sequence in E is the same.

A related nuance is due to hardware interrupts. If P_1 experiences a hardware interrupt and P_2 does not, we make the safe choice: assume the program states are not equal. (It is easy to make the opposite choice, by simply ignoring hardware interrupts.)

7 EXPERIMENTAL EVALUATION

We now describe **CHEX**'s implementation and present an extensive evaluation of **CHEX** for multiversion replay.

Implementation. **CHEX** is implemented in C and Python. **CHEX** relies on Sciunit [1] for monitoring the application on Alice's side and it relies on Checkpoint/Restore in Userspace (CRIU) [10] to checkpoint/restore program states. **CHEX** maintains a ramfs cache to maintain checkpoints. These checkpoints are of the process corresponding to the REPL program and not of the container that Sciunit creates.

We use CRIU as a checkpointing mechanism. This is precisely to enable checkpoint of a process independent of its programming language[3]. CRIU does not freeze the state of the container but just the application process. Currently, **CHEX** is integrated with the IPython kernel. In future, we plan to integrate **CHEX** with Xeus [9], which will help us extend **CHEX** to C programs as well. For the purposes of reproducibility we have made available the code for the audit and replay mode of **CHEX** at [40].

We used a combination of real-world applications and synthetic datasets for evaluation. We ran all our experiments on a 2.2GHz Intel Xeon CPU E5-2430 server with 64GB of memory, running 64-bit Red Hat Enterprise Linux 6.5. The heuristics were developed in Python 3.4.

Real-world Applications. We searched GitHub and identified compute- and data-intensive notebooks, i.e., the programmer had already divided the code into cells. Most of these notebooks were published as artifacts in specific domain conferences (pre-established to be reproducible), and they were described as compute- and data-intensive.

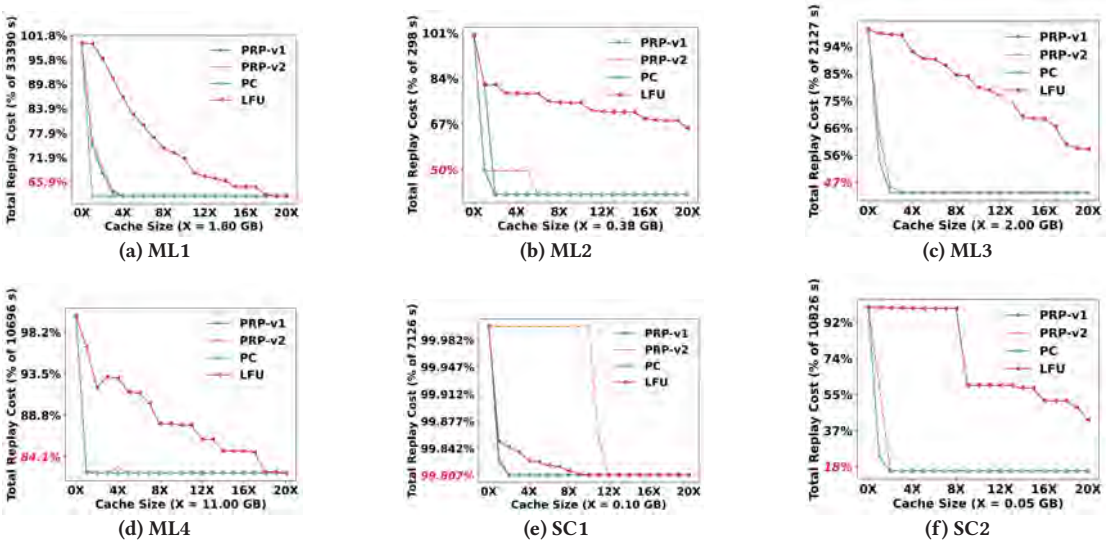
We used four neural network machine learning applications (**ML**) and two scientific computing (**SC**) applications. Table 1 describes the characteristics of these notebooks. For the majority of the applications, *the number of versions* was determined in consultation with the notebook authors, by identifying meaningful changes to parameter values. Other notebooks were changed similarly. *Total replay cost* is the time to run all the versions with no cache. *Total checkpoint size* is the space required if each corresponding cell of the execution tree is checkpointed. *Cell compute range* and *Cell checkpoint size* represents the range of cell compute time and checkpoint size ranges, respectively. The *changed parameter* row mentions application parameters that were changed to create versions. The only way we created versions was by changing parameters. We did not modify any other part of the programs.

The case of the parameter *epochs* in **ML** notebooks is special. In our case, the **ML** notebooks embed deep neural networks, in which typically the compute-intensive part is the back propagation during the training phase. Back propagation is usually implemented as an iterative for-loop, whose upper bound is defined by the *epochs* parameter. Changing *epochs* will change the training length and the number of iterations in the for-loop. Such a change to create a new version, however, will also re-run the entire training phase again,

[3]Native serializations, *viz.* Pickle, provide only a slight performance benefit (1-2%).

Table 1: Six Real-world Applications

Dataset:	ML1	ML2	ML3	ML4	SC1	SC2
Description	Neural Networks [49, 50]	Stock Prediction [27, 28]	Image Classification [33]	Time-Series Forecast [13]	Gas Market Analysis [2]	Spatial Analysis [45, 46]
Changed parameter	models, hyperparameters, test metrics, datasets, epochs				datasets and input parameters	
Number of versions	25	24	32	36	12	23
Version Length	9 - 13	9	7 - 8	17	18	33
Total (no-cache) replay cost (s)	33390	298	2127	10696	7126	10826
Cell compute range (s)	0.0005 - 1073	0.0003 - 8.5	0.008 - 50	0.01 - 240	0.0003 - 926	0.0002 - 224
Total checkpoint size (GB)	57	37	106	566	13	14
Cell checkpoint size (GB)	0.2 - 1.8	0.2 - 0.38	0.4 - 2	1.3 - 11	0.077 - 0.100	0.040 - 0.050



(a) ML1 (b) ML2 (c) ML3

(d) ML4 (e) SC1 (f) SC2

Figure 8: Performance of DFS algorithms on 6 real-world applications. X denotes the size of the largest checkpoint cell as specified in the last row of Table 1. The y-axis is truncated to show finer performance variations between algorithms.

Table 2: Three Synthetic Datasets

Dataset:	CI	DI	AN
Max. Branch-out Factor	4	4	4
Max. Version Length	6	6	6
Number of versions	20	20	20
Total (no-cache) replay cost (s)	~20000	~5000	~20000
Cell compute range (s)	100 - 600	100	100 - 600
Total storage size (GB)	~22	~18	~18
Cell checkpoint size (GB)	0.5	0.1 - 0.6	0.1 - 0.6

which will include the training iterations performed in the previous version. Therefore to create a version when the change is to *epochs*, we do not modify the value in-place. Instead we add a new cell. This cell consists of the author-provided training loop but with a incremental range of epochs starting from the last epoch value of the previous cell. This way of modifying the epoch parameter introduces no change to the code and corresponds to incremental training, which is often used in ML to take advantage of previous computations.

Synthetic. To test the sensitivity of our heuristics we randomly generated synthetic execution trees with different costs and sizes. We controlled the tree structure using the following parameters:
max. branch out factor: The maximum number of branches possible at a node. Each branch is constructed with a 50% probability. This leads to trees in which many nodes have a single child. This is what we have observed in real notebooks.
max. version length: The number of cells in each version. In general, the length for each version is different because of the randomization described above.
max. number of versions: The number of leaves in the execution tree generated by using *max. branch out factor* and *max. version length*. Using the above parameters, we generate three synthetic datasets:

- Compute intensive (**CI**): In the **CI** tree, the compute cost (δ) of cells is high and the checkpoint cost (sz) is modest.
- Data intensive (**DI**): In the **DI** tree, the checkpoint cost (sz) of cells is high and compute cost (δ) is modest.
- Analytic (**AN**): In the **AN** tree, compute and checkpoint costs *i.e.*, δ and sz increase with *version length*.

Table 2 presents the total compute time and total storage size as well as the compute and storage ranges per cell.

Baselines. IncPy [17, 18] avoids recomputing a function with the same inputs when it is called repeatedly or across program versions. Despite our best attempts we could not get IncPy to run with our real datasets. IncPy is not longer actively maintained and is Python 2.7 based which creates conflicts with more recent notebooks. We simulated the Vizier system, by taking one notebook version at a time [6], and using the simple caching policy that is used for Vizier: Least Frequently used (**LFU**), which is a standard caching algorithm. We adapt **LFU** to our case by checkpointing every cell of the first version of a notebook till the cache space fills up. As subsequent versions arrive, the cache eviction policy is decided by the measure $frequency \times \#\ of\ nodes\ in\ subtree/cell\ size$, i.e., retaining cells which are used frequently and are responsible for larger subtree, normalized by their size. Least recently used, another standard caching algorithm, is not relevant in our case due to the depth-first replay order.

7.1 Experiments

We first evaluate the benefit different algorithms provide in terms of reduction in replay time. We then evaluate the overhead of operating **CHEX**.

7.1.1 **Comparing decrease in replay cost via different algorithms**. Persistent Root Policy (**PRP**) and Parent Choice (**PC**) make different choices with respect to cells that must be retained in cache for recomputation. In this experiment, we evaluate how those decisions compare with the (**LFU**) baseline. Recall that **PRP** has two versions: **PRP-v1**, in which we cache checkpoints greedily based on contribution to reduction in cost, and **PRP-v2**, in which we normalize the cost reduction by the checkpoint size.

To compare algorithmic performance, we choose a cache size that is equal to the largest checkpoint size in a notebook and compute total replay time. The y-axis is initialized with a non-zero value to show finer comparisons between algorithms. For both **PC** and **PRP** algorithms on real-world applications, as is expected, Figures 8(a)-(f), show decreasing compute times (y-axis) as the cache size is increased (x-axis). We also see **PRP** and **PC** always perform substantially better than **LFU**, and **PC** reduces total compute cost more than either of the **PRP** versions.

Both these result trends are not exhibited in Figure 8(e) (**SC1**) and, to an extent, in Figure 8(d) **ML4**. In (e), as we observe, none of the algorithms, including the baseline **LFU**, show any benefit of caching. This is because in this notebook only the last cell of each version is compute-intensive, and none of the intermediate cells are cache-worthy. In (d), similarly, most computation is towards the later cells; **PRP** and **PC** still find some ways to optimize which **LFU** cannot find. The effect of reuse of intermediate results is well-demonstrated when comparing **ML4** and **SC2** which exhibit similar total replay costs. However, there is a much greater reduction in total replay cost in **SC2** (from 100% to 18%) as there are several compute-intensive pre-processing steps in the earlier cells of the notebook, where as in **ML4** most computation occurs towards the later cells.

Analyzing deeper we also observe these trends: (i) Sometimes, initially, **PRP** performs better, and this happens due to small cache size effect, since **PC** becomes a clear win with some additional cache space; (ii) **PRP-v1** performs better than **PRP-v2** indicating that eviction on a cost/size measure leads to more greedy eviction policy where checkpoints are evicted which need to recomputed later; and finally (iii) **ML1**, **ML3** and **SC2** are compute-intensive notebooks. Using the **PC** algorithm, these notebooks show a reduction of 60-65% in their compute time at a size of the cache which is at most double the size of the largest checkpoint cell in the notebook. This indicates that smart algorithms can provide significant benefits even with small cache sizes. We obtain similar results for synthetic datasets, and, for lack of space, only include the figures for synthetic results in the extended version of the paper [34].

7.1.2 **Determining number of versions replayed with fixed cache size.** We also examine the direct benefit of a system like **CHEX** for users. For most users **CHEX** will be configured with a given amount of cache space. Users, however, have time constraints. Thus we determine, for given cache sizes, number of versions that can be replayed with **CHEX** in a given amount of time, on the **AN** dataset. Figure 9(a) presents the result (number of versions (y-axis) for the amount of time it takes to replay them (x-axis)) for a given cache size, the value being either: no cache, 0.25GB, 0.5GB, and 1GB. The Figure shows that a user can run 50% more number of versions by doubling the space for the same fixed amount of time. To be able to run larger number of runs for the same amount of time has implications for scaleable collaborative sharing and artifact evaluation use cases.

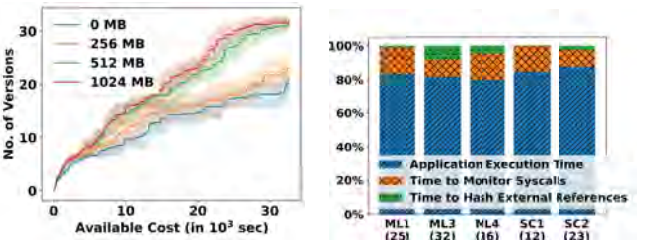


Figure 9: (a) For given cache sizes, number of versions that can be replayed with CHEX in a given amount of time, on the AN dataset. (b) The overhead of auditing δ, sz, g and h in real world applications with > 5 minutes of replay cost.

7.1.3 Time and space required to run **CHEX**. We first determine the cost of auditing an application in **CHEX**.

Cost of Auditing. **CHEX** performs auditing of state for each version of an application in terms of computation time δ, state size sz, state code hash h, and state lineage g. We report both normal execution and audited execution as a percentage of the total time of using **CHEX** on a real application.

Amongst these audited quantities, the primary overhead is the additional time required to audit the application for state lineage, i.e., g. We further divide time to audit for g into time required to (i) monitor and log system events in the application, and (ii) the time required to compute the hash of any external content

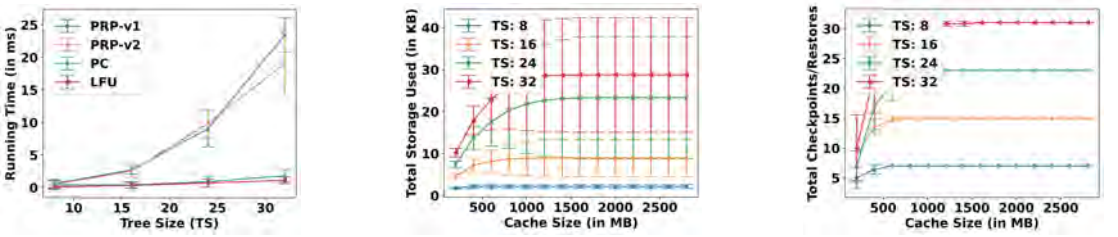


Figure 10: Algorithm complexity of AN workloads: (a) running time, (b) storage size for PC, and (c) number of checkpoints/restore-switch for PC

that is referenced. As Figure 9(b) shows, a 15-25% of total auditing overhead is added across all applications. We are reporting 5 out of six applications as **ML2** has relatively insignificant running time to begin with.

Also the time to perform cell equality and construct the execution tree is negligible. Alice shares a package with the execution tree, the size of which is less than 1KB.

Cost of computing cache eviction decisions. We have shown the multiversion replay problem to be NP-hard; **PRP** and **PC** are heuristic algorithms, and thus have some time and space cost of making the cache eviction decisions. In this experiment, we measure the cost of using **PRP** and **PC** algorithms in comparison to LFU in terms of running time, space used, and number of times checkpoint/restore call was made. We experimented with the **AN** synthetic dataset.

The variability in running time of the algorithms with cache size is negligible (~[0.05%]). Therefore, we fix the cache size to 1GB and show the variation with respect to two other parameters, the number of nodes in the tree, i.e., tree size and the different algorithms. Figure 10 (a) shows that **PC** is better than **PRP** in terms on run-time overhead, but in terms of space, it does incur a cost.

PRP has negligible state maintenance as it uses the execution tree to determine the order. However, **PC** takes storage, because it has to store all possible combinations of execution orders for different cache eviction sizes to get the most optimal one. Figure 10 (b) shows the increase in storage for different tree sizes as cache size is increased. Despite these differences, we highlight that the runtime and memory overheads of both algorithms is much lower (0.5-2%) than the overall compute time and storage of multiversion execution of any given real dataset.

The above experiment measures decision-making time and space. In practice to implement the decisions we must account for in-memory checkpoints and restore (C/R) time. In general, time to C/R are proportional to the size of the checkpointed state and are negligible. So we measured the number of times C/R were performed to check if small C/R costs adds to the overall latency of multiversion execution (Figure 10 (c)). As we see C/R costs are negligible, and decision making accounts for the primary cost.

Apart from the experiments reported above, we attempted a comparison between **PC** and an optimal algorithm,using the **AN** dataset for comparison. For optimal, we wrote our problem, the MVR-P, as an Integer Linear Program (ILP) and attempted to solve it with the Couenne optimizer [30]. The Couenne timeout was set as ten minutes. For tree size of 2-6 nodes, Couenne finished finding a solution in less than 10 seconds, but after that the time starts increasing exponentially. At 12 versions and an execution tree of 20 nodes, the optimal solution could not be found within the set time out. On increasing the number of versions, it took more time to find the optimal solution than naive replay (without cache). On the other hand, as we show in Figure 10(a) we took milliseconds to find a solution for more than a tree-size of 30.

Since we only found optimal solution for small trees, in terms of the quality of the solution, we found the replay cost of **PC** similar to optimal. For larger tree sizes, it may give better cost estimates, but given the large running time of optimal for larger and complex instances, we assess, it is not worth it. Finally, the overhead of implementing the decisions in **CHEX** is too small to be measured and often smaller than the variance between multiple runs.

8 RELATED WORK

Tools and hubs for sharing and reuse. Sharing and replaying is essential for verifying, and reproducing complex applications. Several user-space virtualization based tools have recently been proposed to enable sharing and repeating computations [8, 19, 22, 41, 43, 55]. These tools do not address multiversion replay. In a virtualization package, code and data remain separate as files or databases [43]. Computational notebooks, which combine code and data, have received wide attention recently for sharing and use [26]. Notebook sharing, like package sharing, is easy but *(re-)*execution across versions remains sequential. Nodebooks [58] and Vizier [6] are specialized notebook clients that support and store notebook versions at a cell level. Neither, however, compute deltas between versions or trade computation for storage. Our work complements specialized notebook systems used for interactive development [24], and given lineage from these systems [31], replay can be enabled.

Execution lineage. There are several provenance models for capturing execution lineage [52]. In this paper, we adopt the system-event trace analysis process that is also used in other whole system provenance tracking methods [3, 15, 51].

Data caching. Data management systems have a rich history of employing object caches that tradeoff space for time to improve performance of applications. Semantic caching allows caching of query results [12, 48], web-object caching allows caching of web objects [7, 23], and query-based object caching allows database object caching based on queries [32]. In all of these works, the workload sequence is not known. In the multi-query scenarios [48] the workload is presented as set of queries and hence there is the possibility of caching the results of common sub-expressions and reusing them

across queries. However, efficient reuse in the multi-query setting primarily involves searching through the space of query answering plans to identify plans that could potentially lead to optimal reuse. In certain cases not finding the optimal plan and blindly reusing common subexpressions may blow up the computation time because a large join may be required. Our scenario appears similar but we do not have the wiggle room provided by the semantics of a query, nor the potential pitfalls associated with blind reuse.

State management for recomputation. [54] provides an excellent survey of state management for computation. State can be recomputed from lineage or state can be stored 'as-is'. In SciInc [56] state is recomputed from lineage that is versioned. Versioned lineage or causality-based versioning [36, 56] leads to correct computation of state for incremental replay. In this work, on the contrary, we are concerned with state that is stored 'as-is'. Several works store 'as-is' state—this state is state of a variable, query, program, or configuration [54]. Similar to [20, 21, 25, 39], in this work, our operator is program state. However, in these works the purpose is fault-tolerance, and so the system periodically checkpoints but does not consider space limitations. We determine a limited number of checkpoints of program state to save in-memory space, and using lineage, choose to simply recompute when efficient. To reduce space an alternative would be to incrementally checkpoint as explored in differential flows [35, 37] and query re-optimization [29]. These approaches are not extendable to checkpoints of program state, which is an in-memory map. Very recently checkpointing was used to improve efficiency, but the checkpoint frequency is periodic [14].

Checkpoint location. Deciding when to checkpoint has received attention in HPC scheduling [5, 47]. A primary objective is to minimize the amount of computation that needs to be redone in case the system fails. In HPC workflows, the checkpoint also has an overhead. We consider machine learning and scientific computing programs in which the checkpoint overhead is nearly zero.

Closer in spirit to our work is the DataHubs [4] system that seeks to maintain multiple versions of large data sets without fully replicating them. In this system some versions are stored fully materialized and others are stored only as deltas linked to other versions. The problem is to trade off total storage required versus time taken to recreate a version. At a glance, it is possible to think that the program states of the cells of our multiversion program can be aligned with the data sets considered in DataHubs. However, the fundamental difference is that DataHubs assumes each version of a data set has *already been created* the first time. Thus, they assume that at least one version of the data set is stored in its entirety. In **CHEX**, the equivalent thing would be for Alice to share some of the program states generated in her execution with Bob. This defeats the entire purpose of independent repetition by Bob.

9 DISCUSSION

We now discuss any assumptions that **CHEX** makes and our results. We assumed that **CHEX** works with REPL cells, but, in general, we do not constrain users like Alice to program with REPL interfaces. If the code is not developed via a REPL interface, **CHEX** preprocesses it into cells, akin to a program developed via a REPL interface, before monitoring. This preprocessing takes care to not split functions or control flows into separate cells. Thus every input program is automatically transformed into an equivalent REPL program and then entered into the **CHEX**.

We have assumed multiple versions for a given program. We make no assumptions on the types of edits that constitutes a version on Alice's side. Thus, Alice can change values of parameters, specifications of datasets, models, or learning algorithms. She can also add or delete entire cells. In practice we have found such versions to not correspond to development versions but as separate branches in version-control repositories. In workflow systems they also correspond to independent, but related, experiments.

We have only demonstrated a scenario in which Alice shares notebooks with Bob for multiversion replay. A more evolved back-and-forth sharing of packages, one that accounts for any previous multiversion replay decisions to be persisted, will require further changes both to the system and the algorithm. In such a scenario, if the caches persist, some intermediate results are available for free and the algorithm needs to accommodate for that accordingly. This scenario is part of our future work.

Finally, our experiments show that **CHEX** significantly decreases the replay time for notebooks and allows a user to execute a far higher number of versions in a given amount of time. The benefit arises particularly for notebooks where pre-processing or training steps are compute and data-intensive. In particular, if all computation is conducted in the last cell, then opportunities for optimization on intermediate results reduce drastically. In this case, one option is to encourage the developer to further divide the last cell, which creates further opportunities of optimization. If the cell cannot be divided, then one may employ a hybrid approach of using function-based caching within this cell. This may, however, require some analysis of the program in the last cell.

10 CONCLUSION

In this work we have highlighted the need for improving the efficiency of multiversion replay. Our work shows that execution lineage can be used to establish cell equality and reuse shared program state to optimize replaying of multiversions. We show that optimizing is not trivial and, given a fixed cache size, MVR-P is NP-hard and present two efficient heuristics for reducing the total computation time. We develop novel checkpoint-based caching support for replaying versions and show that **CHEX** is able to reduce the compute time of several machine learning and scientific computing notebooks using a cache size that is smaller than the checkpoint size of a notebook.

In the future, we wish to extend **CHEX** for queries and the standard database provenance model. This problem seems akin to how we previously extended provenance-based application virtualization [42] to database virtualization [43]. We also wish to explore how **CHEX** can incorporate program restructuring, which happens during interactive notebook development leveraging recent provenance models developed in this area [6, 24, 31] and developing corresponding online algorithms.

ACKNOWLEDGMENTS

This work is supported by National Science Foundation under grants CNS-1846418, NSF ICER-1639759, ICER-1661918 and a Department of Energy Fellowship.

REFERENCES

[1] 2017. Sciunit. https://sciunit.run/. [Online; accessed 10-Sep-2021].

[2] Bahuisman. 2018. Natural-Gas-Model. https://github.com/bahuisman/NatGasModel. [Online; accessed 10-Dec-2021].

[3] Nikilesh Balakrishnan, Thomas Bytheway, Ripduman Sohan, and Andy Hopper. 2013. OPUS: A Lightweight System for Observational Provenance in User Space. In *5th USENIX Workshop on the Theory and Practice of Provenance (TaPP 13)*. 1–4.

[4] Souvik Bhattacherjee, Amit Chavan, Silu Huang, Amol Deshpande, and Aditya Parameswaran. 2015. Principles of dataset versioning: exploring the recreation/storage tradeoff. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1346–1357.

[5] Mohamed-Slim Bouguerra, Denis Trystram, and Frédéric Wagner. 2012. Complexity analysis of checkpoint scheduling with variable costs. *IEEE Trans. Comput.* 62, 6 (2012), 1269–1275.

[6] Michael Brachmann, William Spoth, Oliver Kennedy, Boris Glavic, Heiko Mueller, Sonia Castelo, Carlos Bautista, and Juliana Friere. 2020. Your notebook is not crumby enough, REPLace it. In *Conference on Innovative Data Systems Research (CIDR)*.

[7] Pei Cao and Sandy Irani. 1997. Cost-aware www proxy caching algorithms.. In *USENIX Symposium on Internet Technologies and Systems*. 193–206.

[8] Fernando Chirigati, Rémi Rampin, Dennis Shasha, and Juliana Freire. 2016. ReproZip: Computational Reproducibility With Ease. In *SIGMOD'16*. 2085–2088.

[9] Jupyter Community. 2016. C++ implementation of the Jupyter Kernel protocol. https://github.com/jupyter-xeus/xeus.

[10] The CRIU Community. 2019. Checkpoint/Restore In Userspace. https://criu.org/. [Online; accessed 8-Jan-2019].

[11] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press, Chapter 15.

[12] Shaul Dar, Michael J Franklin, Björn Þór Jónsson, Divesh Srivastava, and Michael Tan. 1996. Semantic Data Caching and Replacement. In *Proceedings of the 22th International Conference on Very Large Data Bases*. 330–341.

[13] Joseph Eddy. 2019. Time-Series Forecasting. https://github.com/JEddy92/TimeSeries_Seq2Seq. [Online; accessed 10-Dec-2021].

[14] Rolando Garcia, Eric Liu, Vikram Sreekanti, Bobby Yan, Anusha Dandamudi, Joseph E Gonzalez, Joseph M Hellerstein, and Koushik Sen. 2020. Hindsight logging for model training. *Proceedings of the VLDB Endowment* 14, 4 (2020), 682–693.

[15] Ashish Gehani and Dawood Tariq. 2012. SPADE: Support for Provenance Auditing in Distributed Environments. In *Proceedings of the 13th International Middleware Conference (Middleware '12)*. 101–120.

[16] Pradeep Kumar Gunda, Lenin Ravindranath, Chandu Thekkath, Yuan Yu, and Li Zhuang. 2010. Nectar: Automatic Management of Data and Computation in Datacenters. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI)*. 75–88.

[17] Philip Guo and Dawson Engler. 2011. Using Automatic Persistent Memoization to Facilitate Data Analysis Scripting. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA '11)*. ACM, 287–297.

[18] Philip J. Guo and Dawson Engler. 2010. Towards Practical Incremental Recomputation for Scientists: An Implementation for the Python Language. In *Proceedings of the 2nd Workshop on the Theory and Practice of Provenance (TAPP'10)*. 6–6.

[19] Philip J. Guo and Dawson Engler. 2011. CDE: Using System Call Interposition to Automatically Create Portable Software Packages. In *2011 USENIX Annual Technical Conference (USENIX ATC 11)*. 21–21.

[20] Doug Hakkarinen and Zizhong Chen. 2012. Multilevel diskless checkpointing. *IEEE Trans. Comput.* 62, 4 (2012), 772–783.

[21] Jeong-Hyon Hwang, Ying Xing, Ugur Cetintemel, and Stan Zdonik. 2007. A cooperative, self-configuring high-availability solution for stream processing. In *2007 IEEE 23rd International Conference on Data Engineering*. IEEE, 176–185.

[22] Yves Janin, Cédric Vincent, and Rémi Duraffort. 2014. CARE, the Comprehensive Archiver for Reproducible Execution. In *Proceedings of the 1st ACM SIGPLAN Workshop on Reproducible Research Methodologies and New Publication Models in Computer Engineering (TRUST '14)*. 1–7.

[23] Shudong Jin and Azer Bestavros. 2000. Popularity-aware greedy dual-size web proxy caching algorithms. In *Proceedings 20th IEEE International Conference on Distributed Computing Systems*. IEEE, 254–261.

[24] David Koop and Jay Patel. 2017. Dataflow notebooks: encoding and tracking dependencies of cells. In *9th USENIX Workshop on the Theory and Practice of Provenance (TaPP 2017)*. 17–17.

[25] YongChul Kwon, Magdalena Balazinska, and Albert Greenberg. 2008. Fault-tolerant stream processing using a distributed, replicated file system. *Proceedings of the VLDB Endowment* 1, 1 (2008), 574–585.

[26] Sam Lau, Ian Drosos, Julia M Markel, and Philip J Guo. 2020. The Design Space of Computational Notebooks: An Analysis of 60 Systems in Academia and Industry. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 1–11.

[27] Xinyi Li. 2020. Stock Prediction Using Financial News. https://github.com/AI4Finance-LLC/Financial-News-for-Stock-Prediction-using-DP-LSTM-NIPS-2019. [Online; accessed 5-Dec-2021].

[28] Xinyi Li, Yinchuan Li, Hongyang Yang, Liuqing Yang, and Xiao-Yang Liu. 2019. DP-LSTM: Differential privacy-inspired LSTM for stock prediction using financial news. *33rd Conference on Neural Information Processing Systems (NeurIPS 2019) Workshop on Robust AI in Financial Services: Data, Fairness, Explainability, Trustworthiness, and Privacy* (2019).

[29] Mengmeng Liu, Zachary G Ives, and Boon Thau Loo. 2016. Enabling incremental query re-optimization. In *Proceedings of the 2016 International Conference on Management of Data*. 1705–1720.

[30] Robin Lougee-Heimer. 2003. Convex Over and Under ENvelopes for Nonlinear Estimation. https://www.coin-or.org/Couenne/. [Online; accessed 21-July-2021].

[31] Stephen Macke, Hongpu Gong, Doris Jung-Lin Lee, Andrew Head, Doris Xin, and Aditya Parameswaran. 2021. Fine-grained lineage for safer notebook interactions. *Proceedings of the VLDB Endowment* 14, 6 (2021), 1093–1101.

[32] Tanu Malik, Randal Burns, and Amitabh Chaudhary. 2005. Bypass caching: Making scientific databases good network citizens. In *21st International Conference on Data Engineering (ICDE'05)*. IEEE, 94–105.

[33] Nithin Manne. 2020. Image Classification. https://www.kaggle.com/nithinmanne/fashionmnist. [Online; accessed 10-Dec-2021].

[34] Naga Nithin Manne, Shilvi Satpati, Tanu Malik, Amitabha Bagchi, Ashish Gehani, and Amitabh Chaudhary. 2022. CHEX: Multiversion Replay with Ordered Checkpoints. arXiv:2202.08429 [cs.DB]

[35] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. 2013. Differential Dataflow. In *CIDR*.

[36] Kiran-Kumar Muniswamy-Reddy and David A. Holland. 2009. Causality-based Versioning. *Transactions of Storage* 5, 4 (Dec. 2009), 1–28.

[37] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 439–455.

[38] Yuta Nakamura, Tanu Malik, and Ashish Gehani. 2020. Efficient Provenance Alignment in Reproduced Executions. In *12th International Workshop on Theory and Practice of Provenance (TaPP 2020)*. 6–12.

[39] Bogdan Nicolae and Franck Cappello. 2013. AI-Ckpt: leveraging memory access patterns for adaptive asynchronous incremental checkpointing. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*. 155–166.

[40] Tanu Malik Nithin Naga Manne. 2021. The CHEX System. https://bitbucket.org/depauldbgroup/storagevscompute/src/optimal/.

[41] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. 2017. Engineering record and replay for deployability. In *2017 USENIX Annual Technical Conference*. 377–389.

[42] Quan Pham, Tanu Malik, and Ian Foster. 2013. Using Provenance for Repeatability. In *USENIX Theory and Practice of Provenance (TaPP'13)*. Article 2, 2:1–2:4 pages.

[43] Quan Pham, Tanu Malik, Boris Glavic, and Ian Foster. 2015. LDV: Light-weight database virtualization. In *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 1179–1190.

[44] Quan Pham, Severin Thaler, Tanu Malik, Ian Foster, and Boris Glavic. 2015. Sharing and Reproducing Database Applications. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1988–1991. https://doi.org/10.14778/2824032.2824118

[45] Michael Rilee. 2020. STARE Cookbooks: STARE+Dask-Demo. https://bit.ly/37dlK4B. [Online; accessed 10-Dec-2021].

[46] Michael Rilee, Niklas Griessbaum, Kwo-Sen Kuo, James Frew Frew, and Robert Wolfe. 2020. STARE-based Integrative Analysis of Diverse Data Using Dask Parallel Programming. *Proceedings of ACM SIGSPATIAL conference (SIGSPATIAL'20)* (2020), 417–420.

[47] Yves Robert, Frédéric Vivien, and Dounia Zaidouni. 2012. On the complexity of scheduling checkpoints for computational workflows. In *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN 2012)*. IEEE, 1–6.

[48] Prasan Roy, Srinivasan Seshadri, S Sudarshan, and Siddhesh Bhobe. 2000. Efficient and extensible algorithms for multi query optimization. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. 249–260.

[49] Hojjat Salehinejad. 2020. EPruning (EDropout). https://github.com/sparsifai/epruning. [Online; accessed 10-Dec-2021].

[50] Hojjat Salehinejad and Shahrokh Valaee. 2020. EDropout: Energy-Based Dropout and Pruning of Deep Neural Networks. *arXiv preprint arXiv:2006.04270* (2020), arXiv–2006.

[51] Manolis Stamatogiannakis, Paul Groth, and Herbert Bos. 2014. Looking inside the black-box: capturing data provenance using dynamic instrumentation. In *International Provenance and Annotation Workshop*. Springer, 155–167.

[52] Manolis Stamatogiannakis, Hasanat Kazmi, Hashim Sharif, Remco Vermeulen, Ashish Gehani, Herbert Bos, and Paul Groth. 2016. Trade-Offs in Automatic Provenance Capture *(IPAW 2016)*. Springer-Verlag, 29–41.

[53] Victoria Stodden, Matthew S Krafczyk, and Adhithya Bhaskar. 2018. Enabling the Verification of Computational Results: An Empirical Evaluation of Computational Reproducibility. In *Proceedings of the First International Workshop on Practical*

Reproducible Evaluation of Computer Systems. ACM, 3.
[54] Quoc-Cuong To, Juan Soto, and Volker Markl. 2018. A survey of state management in big data processing systems. *The VLDB Journal* 27, 6 (2018), 847–872.
[55] Dai Hai Ton That, Gabriel Fils, Zhihao Yuan, and Tanu Malik. 2017. Sciunits: Reusable Research Objects. In *IEEE eScience*. 374–383.
[56] Andrew Youngdahl, Dai Hai Ton That, and Tanu Malik. 2019. SciInc: A Container Runtime for Incremental Recomputation. In *IEEE eScience*. IEEE, 291–300.
[57] Zhihao Yuan, Dai Hai Ton That, Siddhant Kothari, Gabriel Fils, and Tanu Malik. 2018. Utilizing Provenance in Reusable Research Objects. *Informatics* 5, 1 (2018), 14. https://doi.org/10.3390/informatics5010014
[58] K Zielnicki. 2017. Nodebook. https://multithreaded.stitchfix.com/blog/2017/07/26/nodebook/ [Online; accessed 10-July-2021].