THE HONG KONG
POLYTECHNIC UNIVERSITY
香港理工大學
Pao Yue-kong Library
包玉剛圖書館

Copyright Undertaking

This thesis is protected by copyright, with all rights reserved.

By reading and using the thesis, the reader understands and agrees to the following terms:

1. The reader will abide by the rules and legal ordinances governing copyright regarding the use of the thesis.
2. The reader will use the thesis for the purpose of research or private study only and not for distribution or further reproduction or any other purpose.
3. The reader agrees to indemnify and hold the University harmless from and against any loss, damage, cost, liability or expenses arising from copyright infringement or unauthorized usage.

IMPORTANT

If you have reasons to believe that any materials in this thesis are deemed not suitable to be distributed in this form, or a copyright owner having difficulty with the material being included in our database, please contact lbsys@polyu.edu.hk providing details. The Library will look into your claim and consider taking remedial action upon receipt of the written requests.

Pao Yue-kong Library, The Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong

http://www.lib.polyu.edu.hk

Speeding Up Querying and Mining Operations on Data with Modern Hardware

Fang Wang

PhD

The Hong Kong Polytechnic University

2022

The Hong Kong Polytechnic University

Department of Computing

Speeding Up Querying and Mining Operations on Data with Modern Hardware

Fang Wang

A thesis submitted in partial fulfilment of the requirements

for the degree of Doctor of Philosophy

November 2021

CERTIFICATE OF ORIGINALITY

I hereby declare that this thesis is my own work and that, to the best of my knowledge and belief, it reproduces no material previously published or written, nor material that has been accepted for the award of any other degree or diploma, except where due acknowledgement has been made in the text.

_____(Signed)

Fang Wang (Name of student)

Abstract

In recent years, the rapid expansion of variety, velocity, and volume of data leads to various challenges on efficiency of querying and mining data. In this thesis, we identify three challenging problems on querying and mining data, and propose optimized solutions by exploiting modern hardware like GPUs and emerging non-volatile memory.

Data mining enables us to discover hidden knowledge from data. The past decades have witnessed the great successes of data mining in many applications such as bioinformatics and software engineering, business intelligence, and search engines. Similarity computation is a core subroutine of many mining tasks on multi-dimensional data, which are often massive datasets at high dimensionality. However, the ever-expanding volumes and dimensionality of data lead to similarity computation being the bottleneck that prolongs the process of mining. In these mining tasks, the performance bottleneck is caused by the memory wall problem as a substantial amount of data needs to be transferred from the memory to processors. Recent advances in non-volatile memory (NVM) based processing-in-memory (PIM) enjoy the ability to process the data without moving them out of memory, which can reduce data transfer and thus alleviate the performance bottleneck of the mining tasks. Nevertheless, NVM PIM supports specific operations only but not arbitrary operations. We tackle the challenge and carefully exploit NVM PIM to accelerate similarity-based mining tasks on multi-dimensional data without compromising the accuracy of results. Ex-

perimental results show that our proposed method achieves up to 11.0x speedup for representative mining algorithms such as kNN classification and k-means clustering.

Blockchains are distributed systems that provide decentralized, secure, and shared data access among untrusted parties. They have been used in applications such as banking, supply chain, healthcare, and IoT scenarios. New data such as transactions are recorded into a block in the append-only (and immutable) manner. Blockchain maintains a linked list of blocks and grows by mining new blocks. However, the mining process consumes significant computational overhead and easily prolongs the progress of data storage. This is because the mining processing requires the validity of new data to be verified through consensus mechanism - proof-of-work, which expends computational effort solving an arbitrary mathematical puzzle. To improve the data storage performance, we propose a NVM PIM architecture to accelerate blockchain mining. NVM PIM can directly process data at the memory arrays. The large number (e.g., dozens of thousands) of memory arrays of NVM release massive parallelism, and thus is promising to speed up blockchain mining that demands expensive computation resources. We utilize matrix transformation to map the operations in blockchain mining into the matrix multiplication operation, which is supported by NVM PIM. We further propose an intra-transaction and inter-transaction parallel framework to make full use of the parallelism of NVM PIM. The experimental results show that our proposed method outperforms CPU-based and GPU-based implementations significantly.

Analytical query processing is an important function in data warehouses, for systematical data reporting and analysis. Efficient query processing is significant to support sound and timely strategic decisions in todays competitive, fast-evolving enterprise industry. However, with the increase of data volume and complexity of analysis scenarios in real applications, a query with joining multiple relations can easily cost hours and even days. Cardinality estimation estimates the size of the

intermediate result relations. The query processing relies on the estimated cardinalities to evaluate the costs of the execution plans and can find the optimal execution plan if the estimations are error-free. Deep learning has shown attractive effectiveness to provide more accurate estimation than traditional methods. Nevertheless, learning-based estimators consume more estimation time since the model inference triggers expensive computation. GPU is a prevalent accelerator for deep learning model inference due to its high parallelism with many cores. We propose a GPU-enabled learning-based progressive cardinality estimator (LPCE) to speed up query end-to-end execution. LPCE runs on GPUs and enjoys both short inference time and high estimation accuracy. In addition, to serve cardinality estimation before query execution, LPCE can progressively refine the estimations during the query execution process. We integrate LPCE into PostgreSQL and conduct extensive experiments on real datasets. The results show that LPCE significantly outperforms existing cardinality estimators in end-to-end query execution time.

In summary, in this thesis, we study how to leverage modern hardware, especially NVM and GPU, to optimize three types of querying and mining operations on data: similarity-based data mining, blockchain mining, and analytical query execution.

Acknowledgements

I would like to thank all the people who gave me tremendous support during my Ph.D. study.

First and foremost, I want to express my deepest sense of gratitude to my supervisors, Dr. Man Lung Yiu and Prof. Zili Shao. I have learned a lot for their incredible guidance and support. It is my great pleasure to be a student of Dr. Yiu. He gave me many elaborate suggestions and guidance, and showed me the right direction but allowing me to walk the path myself. I also want to thank Prof. Shao, who helped me establish the positive attitude to research. Prof. Shao gave me a book named "The road less traveled", which gives me many lifetime lessons.

I would like to thank Dr. Bo Tang and Dr. Xiao Yan at Southern University of Science and Technology. I learned a lot from the discussions and interactions with them in academic research.

Next, I sincerely thank my research colleagues and friends during my Ph.D. study. Many thanks to Zhe Li, Jiahao Zhang, Shuai Li, Zhenlin An, Ningning Hou, Kaiyan Cui, Zexin Lu and Jingcai Guo. Thanks for their considerate assistance on my research and daily life during my Ph.D. study. Another particular thanks goes to my friends in Hong Kong Polytechnic University, Lei Xue, Maria Victorova, Nikolay Lyapunov, Das Kunal Krishna, Pannuzzo Paola, Peichen Wu, and Tani Ryuichi. I am glad to have many joyful moments with all of them.

Finally, my deepest thanks are to my family. Thanks for their endless love,

support, and encouragement through my entire life. They are always on my side, and let me pursue my dream for so long. Also, I am really grateful for Hanyu Deng's companion, encouragement and understanding for the last year of my Ph.D. study. Their love and supports carry me through my Ph.D. study to the end.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Big data in the present era has been exerting an increasingly profound influence on data management. For example, the social network such as Facebook and Twitter produces over 500 TB of data over the time frame of one day [190]. Processing such huge volumes of data brings about extraordinarily complex challenges to the efficient data management. In this thesis, we identify three challenging problems on querying and mining data, and propose optimized solutions with modern hardware such as emerging non-volatile memory (NVM) and graphics processing units (GPUs).

Recently, the NVM techniques such as phase-change memory (PCM) [198] and resistive memory (ReRAM) [9], have been demonstrated the promising potential of processing the stored data in-situ, namely NVM processing-in-memory (PIM) [91, 60]. The efficiency of NVM PIM comes from two-fold: reduction of data transfer and massive data parallelism [21, 140]. First, NVM PIM enables us to directly process the stored data, avoiding data movement from the memory to host processors (e.g., CPU). Second, NVM is composed of a large number (e.g., dozens of thousands) of memory arrays. Each array works as a processing unit that enables concurrent computing. NVM PIM has been applied in many applications including neural network [140, 164, 36], graph computing [169, 221], and DNA alignment [88]. GPU is another prevalent hardware due to its higher computing parallelism and mem-

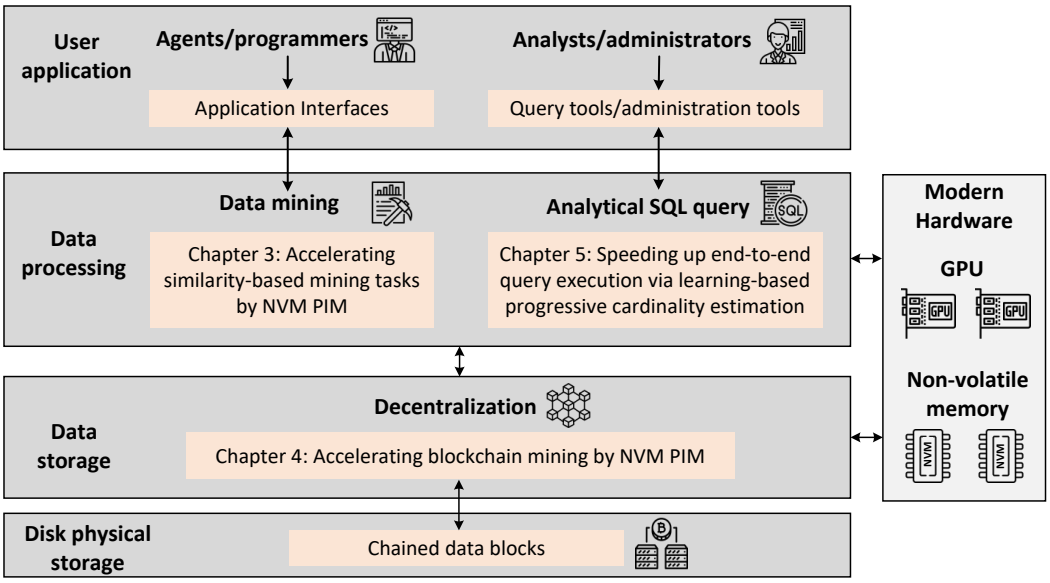


Figure 1.1: Unified research framework.

ory bandwidth than traditional processor - CPU. GPU has been intensively used in compute-intensive applications such as video coding [133, 166], image processing [66, 42].

Figure 1.1 illustrates the overview of our research framework. As shown in the layer of user application, our target users can be programmers and naive users such as web users, and also sophisticated users such as analysts and database administrators. The naive users might invoke the functionality of data mining at the layer of processing through application program interfaces. The analysts and database administrators can utilize query language statements (e.g., SQL) to facilitate analytical query processing. We study to shorten the latency of data mining and analytical query processing by utilizing modern hardware. In addition to data processing, our research also includes the optimization of data storage. We study leveraging the modern hardware to improve the efficiency of blockchain mining process, through which the data at underlying physical storage can enjoy effectively decentralized and secure storage.

Table 1.1: The summary of research problems.

Problem	Targeted platform	Application	Hardware for optimization	Co-processing
Similarity-based data mining (Chapter 3)	Distributed/ centralized system	Bioinformatics, search engine	NVM	✓
Blockchain mining (Chapter 4)	Distributed system	Banking, supply chain	NVM	×
Analytical query execution (Chapter 5)	Distributed/ centralized system	Business analysis	GPUs	✓

Table 1.1 summarizes the three parts of our research framework. In the first part (Chapter 3), we propose to utilize NVM PIM co-processing with host processors to alleviate the performance bottleneck of the data mining tasks. In the second part (Chapter 4), we study utilizing the massive parallelism of NVM PIM to accelerate blockchain mining, which can improve the storage performance of blockchain applications such as banking management. In the third part (Chapter 5), to speed up the analytical query execution, we propose a learning-based cardinality estimator LPCE that efficiently runs on GPUs.

The rest of this chapter is organized as follows. We first briefly the three main components of the unified research framework in Section 1.1 - 1.3. We then give the structure of this thesis and summarize main contributions in Section 1.4.

1.1 Accelerating Similarity-based Mining Tasks by NVM PIM

In this section, we present NVM-based techniques to accelerate similarity-based data mining tasks. Data mining enables us to discover hidden knowledge from data. The past decades have witnessed the great successes of data mining in many applications such as bioinformatics [194, 139], science analysis [71], and search engines [75, 219]. Similarity computation is a core subroutine of many mining tasks on high-dimensional data. Such mining tasks include clustering, classification, motif

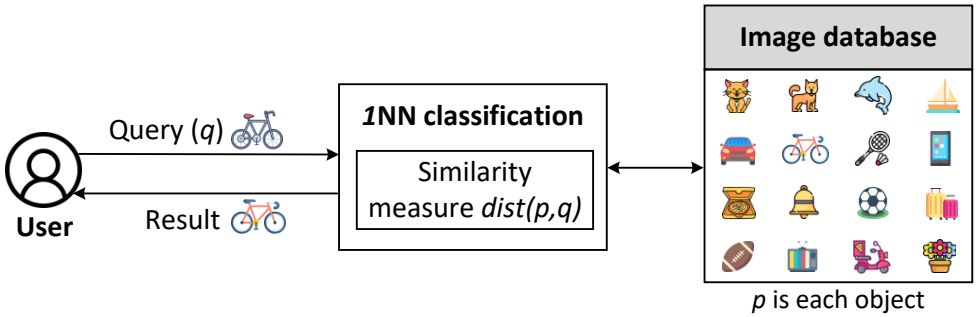


Figure 1.2: Example of similarity-based mining tasks: *1*NN classification.

discovery, and anomaly detection [80]. For instance, Figure 1.2 plots an example of mining tasks in the application of image search - *1*NN classification. These mining tasks often deal with massive datasets at high dimensionality. However, due to the *'memory wall'* challenge caused by the ever-growing gap between processor speed and memory speed [153], the substantial amount of data transferred from the memory to processors in mining tasks becomes the performance bottleneck.

We propose a novel framework to utilize NVM PIM to accelerate a given similarity-based mining algorithm. NVM PIM enables us to process the data at place it resides, which is an ideal approach to address the issue of data transfer. Limited functionality of NVM PIM makes it infeasible to run the entire algorithm. In our approach, NVM PIM works as an assisted processor for similarity computation and the host processors (e.g., CPU) coordinate the remaining operators that incur little data transfer. We design PIM-aware decomposition for a similarity function so that most of its computation can be done by PIM with the supported operations. Moreover, we develop PIM-aware bound function so that it guarantees correct results for the algorithm when the data is floating-point value. The experimental results show that our proposed method achieves up to 11.0x speedup for representative mining algorithms, kNN classification and k-means.

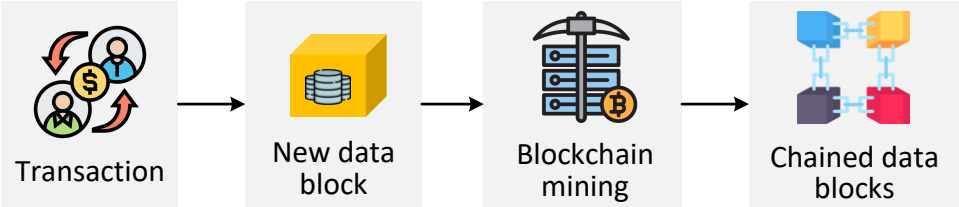


Figure 1.3: Workflow of blockchain-based data storage.

1.2 Accelerating Blockchain Mining by NVM PIM

In this section, we present NVM-based techniques to accelerate blockchain mining, which contributes to the high data storage performance in distributed systems. Blockchain techniques have been intensively used in distributed systems to provide decentralized, secure, and public data management [16]. Blockchain allows shared data access for parties among the untrusted parties in distributed systems [55], which has been used in applications such as banking [74], supply chain [15, 205], healthcare [57, 6] and IoT scenarios [89]. As shown in Figure 1.3, new data such as transactions are recorded into a block in the manner of immutable append-only. Blockchain maintains a linked list of blocks and grows by mining new blocks. However, the mining process consumes significant computational overhead, which causes extremely low data storage performance [55]. For example, the state-of-art blockchain system - Ethereum [199], can only achieve up to 100s of transactions per second [47]. This is because blockchain mining requires the validity of new data to be verified through consensus mechanism - proof-of-work [50], which expends computational effort solving an arbitrary mathematical puzzle [61].

We propose a NVM PIM architecture to accelerate the mining process, to improve the data storage performance of blockchain-based distributed systems. NVM PIM enjoys the massive data parallelism and we offload all the computation of mining into NVM PIM. In order to bridge the gap between the abilities of NVM PIM

and blockchain mining, we design a message schedule module and a SHA computation module. We further devise an intra-transaction parallel framework to accelerate the SHA computation process for each transaction, and an inter-transaction parallel framework to accelerate the blockchain Merkle tree construction and proof-of-work computation process. Experimental results show that the proposed method significantly outperforms up to 778x than CPU-based implementation, and up to 3.8x than GPU-based implementation.

1.3 Speeding Up End-to-end Query Execution Via Learning-based Progressive Cardinality Estimation

In this section, we present fast and accurate techniques to perform cardinality estimation for analytical queries. Analytical query processing is an important function in data warehouses, for systematical data reporting and analysis. For example, through a relational query language (SQL) query, the analyst at a retailer can obtain the report of *"the total sales of the last quarter, grouped by branch"*. Efficient query processing is significant to support sound and timely strategic decisions in today's competitive, fast-evolving enterprise industry. However, with the increase of data volume and complexity of analysis scenarios in real applications, a query with joining multiple relations can easily cost hours and even days [22, 67, 157]. Figure 1.4 presents the end-to-end workflow of query execution. Given a query, cardinality estimation estimates the size of the intermediate result relations. The query optimizer relies on the estimated cardinalities to evaluate the costs of the execution plans, and can find the optimal execution plan if the estimations are error-free [122]. Traditional cardinality estimators fail to account for the correlations among the relations so that suffer from large errors [45, 201, 130, 5].

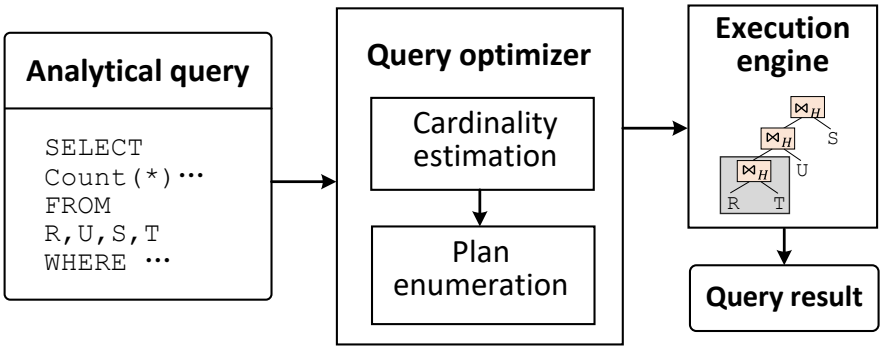


Figure 1.4: Workflow of analytical query execution.

Deep learning has shown the attractive effectiveness to provide more accurate estimation than traditional methods, which can effectively model the joint correlations among tables and extract knowledge from historical query samples [82, 213, 53, 200]. Nevertheless, learning-based estimators consume more estimation time since the model inference triggers expensive computation. GPU is a prevalent accelerator for deep learning model inference due to its high parallelism with many cores [150].

We propose a novel learning-based progressive cardinality estimator (LPCE) to speed up end-to-end analytical query execution. For query processing, GPU is used as assisted processor for efficient cardinality estimation, and host processors (e.g., CPUs) still deal with the query execution on the data. LPCE can progressively refine the cardinality estimations during the query execution, such that the execution plan can be adjusted with more accurate estimates. LPCE consists of two major components: an initial cardinality estimation model and a progressive cardinality refinement model. We integrate LPCE into PostgreSQL [1] and conduct extensive experiments on real datasets. The results show that LPCE significantly outperforms existing learning-based estimators, reducing up to 99.7% of end-to-end query execution time.

1.4 Thesis Organization and Contributions

In this section, we present the organization of this thesis, through which we point out the main contributions.

- In Chapter 2, we present the background knowledge related to the basic techniques throughout the thesis.

- Chapter 3 (based on [193]) studies the acceleration of similarity-based data mining with NVM. We explore utilizing NVM PIM to minimize data movement overhead. We apply performance profiling to identify the performance bottleneck of mining tasks and estimate the potential performance gain of using NVM PIM. We propose PIM-aware decomposition for similarity function so that most of its computation can be offloaded to PIM. We analyze the data compression based on the hardware configuration of NVM, which can avoid the expensive data re-programming on memory.

- Chapter 4 (based on [191]) studies the acceleration of blockchain mining with NVM. For efficient data storage in blockchain distributed systems, we investigate exploiting NVM to improve the performance of the blockchain mining process. We carefully design the matrix transformation to map the complex bitwise operations in mining such as right and left shift, to matrix multiplication supported by NVM PIM. We further devise an intra-transaction and inter-transaction parallel framework to make full use of NVM PIM computational parallelism.

- Chapter 5 (based on [192]) studies the speedup of analytical query execution. We propose a learning-based progressive cardinality estimator, which contributes to fast analytical query execution with better execution plan. We first observe that estimation errors of existing estimator increase with query

complexity and propose progressive cardinality estimation so that the large errors can be detected and amended. To meet the requirements of both accuracy and efficiency, we design the novel techniques of the node-wise loss function, SRU-based model, and knowledge distillation. Moreover, a progressive refinement model is designed to extract information from the executed sub-plans and refine the estimations for the remaining operators. LPCE enjoys both high effectiveness and efficiency of cardinality estimation when runs on GPU.

- In Chapter 6, we present the conclusions of proposed methods and discuss potential future research directions arising from this thesis.

Chapter 2

Literature Review

In this chapter, we review the existing works related to this thesis. Section 2.1 presents the techniques of NVM PIM. Section 2.2 discusses the advances of GPU. Section 2.3 reviews similarity-based data mining tasks. Section 2.4 elaborates blockchain mining. Section 2.5 presents the methods of cardinality estimation for analytical query execution.

2.1 Non-volatile Memory Processing-in-memory

In this section, we first review the development of NVM PIM. We then present the basics of ReRAM, one kind of NVM that has superior characteristics to support efficient processing-in-memory. In this thesis, we focus on exploiting ReRAM PIM to optimize data mining and querying.

2.1.1 Processing-in-memory

Conventional computers are typically based on the Neumann architecture, in which the memory is separated from the processing space, and programs are executed by moving data between the processing units and memory devices, as shown in Figure 2.1. The idea of offloading the computation into memory traces back to

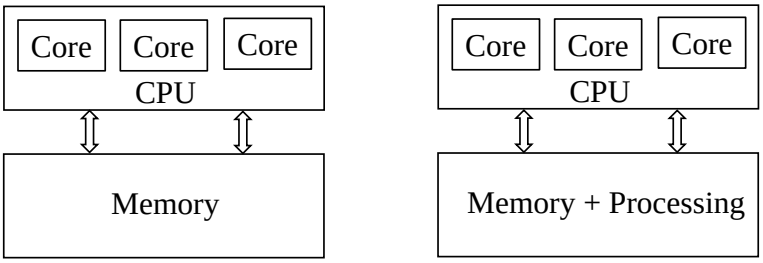


Figure 2.1: Evolution from Neumann architecture (left) to PIM architecture (right).

1970s [65]. In recent years, state-of-the-art memory techniques make it gain extensive attention to accelerate data-intensive applications.

The development of non-volatile memory creates a new horizon for PIM with its capabilities of in-situ analog computation. The efficiency of NVM PIM comes from two sources: *reduction of data transfer* and *massive parallelism* [140, 60]. First, NVM PIM avoids the data transfer between CPU and memory, which always causes high latency and power consumption. Second, massive arrays compose NVMs, and each of them serves as a processing unit to compute multiple data concurrently. Recently, Fujiki et al. [60] designed an NVM-based in-memory processor that enjoys up to 4681 times computational parallelism than modern CPU.

There are several kinds of NVM devices that have been studied to support PIM - spin-transfer torque RAM (STT-RAM) [35], phase-change memory (PCM) [198] and resistive RAM (ReRAM). STT-RAM has the fast read and write speed as reported in Table 2.1, yet the cell size is large and can only store one-bit data. PCM and ReRAM can store multi-bit data in one cell. Compared to PCM, ReRAM is more energy-efficient and provides faster data access.

The various features lead to different supported functions of processing data, as shown in Table 2.2. The most attractive function of ReRAM is vector-matrix computation. [36, 168] are ReRAM-based PIM architectures to accelerate neural network by using the vector-matrix computation. [169, 81, 39] utilize ReRAM crossbars to

Table 2.1: Characteristics of representative NVM techniques [21].

Type	DRAM	ReRAM	PCM	STT-RAM
Volatile	×	✓	✓	✓
Endurance	10^{15}	10^{8}-10^{11}	10^{8}-10^{9}	10^{12}-10^{15}
Read latency (ns)	∼10	∼10	20-60	2-35
Write latency (ns)	∼10	∼50	20-150	3-50
Cell size (F^2)	60-100	4-10	4-12	6-50
Multibit	1	2-7	>2	1
Write energy (J/bit)	10^{-14}	10^{-13}	10^{-11}	10^{-13}

store graphs, and release the massive parallelism to accelerate graph processing. Besides, MAGIC [117] proposes bit-wise operations and multi-bit adders with ReRAM. PCM shares the similar supported functions with ReRAM. [24, 162] investigate the analog vector-matrix multiplications when arrange PCM as a crossbar. [119, 120] extend the applications to transfer learning and compressed sensing. The designs of STT-RAM for PIM focus on bit-wise operations (i.e., AND, XOR) due to its single-bit storage in cells. [96] extends the functions to support basic arithmetic and vector operations. The applications such as text processing, encryption [152] have placed STT-RAM next the host processors to improve computation power.

The superior performance of NVM PIM encourages the exploration of optimizing database. [176] proposes ReRAM PIM approach that allows efficient query processing operations such as projection and aggregation. [92] presents a NVM-based accelerator to perform operations such as MIN, MAX, and Average.

2.1.2 ReRAM Basics

In this thesis, we focus on ReRAM in our PIM design due to its low read latency and energy cost, higher density, and efficient in-situ vector-matrix multiplication. ReRAM has been presented as a promising candidate for future memory systems

Table 2.2: An overview of NVM devices to support PIM.

Types	Functions	Applications
ReRAM	• Vector-matrix multiplication • Logic (i.e., OR, AND) • Arithmetic (i.e., addition, multiplication)	• Deep learning [164, 168, 36, 140] • Graph computing [169, 81, 39] • DNA sequence alignment [88, 77] • Approximate string matching [26]
PCM	• Vector-matrix multiplication • Logic (i.e., NOR, NAND) • Arithmetic (i.e., addition, subtraction)	• Deep learning [162, 161, 24, 23] • Compressed sensing and recovery [120] • Stochastic computing and security [28]
STT-RAM	• Logic (i.e., AND, XOR) • Addition	• Encryption [152] • Convolutional neural network [151]

due to its superior properties [21, 207]. A top electrode, a bottom electrode, and a metal-oxide layer compose the sandwiched structure of ReRAM cell [9], as shown in Figure 2.2(a). It has two switching modes: unipolar and bipolar. The difference of the two modes is switching direction depending on amplitude or polarity of the applied write voltage. Figure 2.2(b) shows the current-voltage curve of a bipolar cell. Applying positive voltage across the cell could switch it from high resistance state (HRS) into low resistance (LRS). Correspondingly, the negative voltage switches the cell from LRS into HRS. LRS and HRS are used to represent logical '1' and '0' respectively. Moreover, in order to support higher storage density, state-of-the-art technologies present multi-bits in single cell (MLC) [147, 208], in which the resistance is changed among multiple levels with finer write control. Recent works reported 1-8 bits precision for each cell [220, 10].

Each cell is connected by two orthogonal nanowires - wordline and bitline. Multiple wordlines and bitlines compose a crossbar [116]. Existing study [147] reported the crossbar size from 4*4 to 1024*1024. Based on Ohm's and Kirchhoff's current law [148], the structure of crossbar makes it natively support dot-product operation: by injecting voltages into wordlines, currents measured at end of bitlines are the dot-

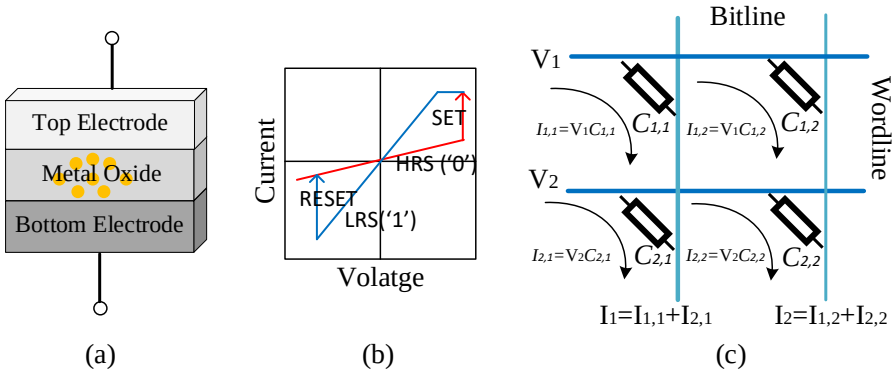


Figure 2.2: Basic of ReRAM: (a) structure of ReRAM cell. (b) current-voltage characteristics of bipolar switching mode. (c) schematic of performing dot-product on crossbar structure.

product results of input voltages and conductances on cells. Figure 2.2(c) depicts an example on 2*2 crossbar, one multiplier V_i is represented as voltages applied into wordlines, and the other multipliers $C_{i,1}$, $C_{i,2}$ are programmed into bitlines [81], current I_1, I_2 sensed at end of bitlines represent $\sum_{i=1}^{2} V_i \cdot C_{i,1}$, $\sum_{i=1}^{2} V_i \cdot C_{i,2}$ respectively.

The efficiency of ReRAM PIM is promising to speed up both memory and compute intensive applications. We discuss more details of ReRAM PIM at the later chapters.

2.2 GPU

In this section, we discuss the design and usage of GPUs. Figure 2.3 shows the overview architecture of GPUs on a modern machine. *Streaming multiprocessor (SM)* is the basic execution unit of GPU, which consists of hardware resources including compute cores and register file. The state-of-art GPUs (e.g, Nvidia A100 GPU [3]) can have up to 80 SMs, and 5012 compute cores. The program function is executed by many threads that are assigned to SMs for processing. One SM can typically hold up to 2048 threads that are organized into *warps*. The threads are executed in the manner of *single instruction multiple threads* (SIMT) model [179], which provide the

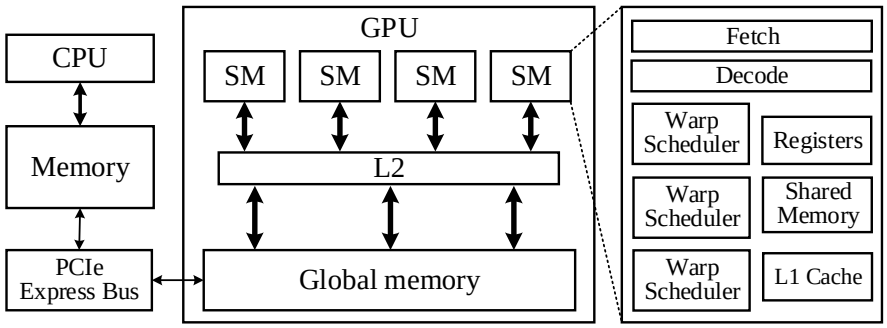


Figure 2.3: GPU architecture overview.

high computing parallelism.

In the memory hierarchy of modern GPUs, *high bandwidth memory* (HBM) [129] is utilized as global memory that can provide 1200 GB/s bandwidth. The on-chip memory such as L2 cache can provide high bandwidth of several TB/s. The capacity of global memory has been steadily scaled up over the past years, achieving up to 32 GB. GPUs typically connect with CPU system through the interconnect bus, such as PCIe. Most modern GPUs adopt PCIe 3.0 [8] that provides up to 16 GB/s bandwidth for data communication between CPU and GPU.

High computing power and memory bandwidth offered by GPU makes it promising to deal with many data-intensive applications. In addition to the successful acceleration efforts on video analysis [133, 166] and image processing [66, 42], GPUs have been extended to support scientific computations such as molecular modeling [173, 11] and DNA sequence alignment [183]. Starting from accelerating single operation including select [165], joins [167, 210] and group by [101, 180], exploiting GPUs for database analytics has been studied for several years. However, the low bandwidth for data communication between CPU and GPU limits the performance benefit on analytical query processing [165].

2.3 Similarity-based Data Mining

In this section, we review the common similarity measures and similarity-based mining algorithms.

2.3.1 Similarity Measure

Distance computation for measuring similarity between objects is an essential subroutine in many mining algorithms. Euclidean distance (ED), cosine similarity (CS), and Pearson correlation coefficient (PCC) are three common measures on floating-point vectors. Besides, binary vector has been an alternative to describe objects in many applications. For instance, image classification tends to compact an image to short binary code using techniques such as locality sensitive hashing (LSH) [30]. Hamming distance (HD) is a common measure on binary codes that preserves the similarity of original objects [98].

Table 2.3: Equation of similarity functions.

Symbol	Equation
ED	$\sum_{i=1}^{d}(p_i - q_i)^2$
CS	$\frac{\sum_{i=1}^{d} p_i q_i}{\sqrt{\sum_{i=1}^{d} p_i^2}\sqrt{\sum_{i=1}^{d} q_i^2}}$
PCC	$\frac{\sum_{i=1}^{d}(p_i-\mu(p))(q_i-\mu(q))}{\sqrt{\sum_{i=1}^{d}(p_i-\mu(p))^2}\sqrt{\sum_{i=1}^{d}(q_i-\mu(q))^2}}$
HD	$\sum_{i=1}^{d}\Delta(p_i - q_i)$ ($\Delta(p_i-q_i)$=0, if p_i=q_i; or 1)

Table 2.3 lists the equations of similarity measures. Let p (resp. q) denote a d-dimensional vector. p_i is the value on i-th dimension. $\mu(p)$ (resp. $\mu(q)$) is the mean value of vector p (resp. q). $dist(p, q)$ denotes the distance between p and q, such as $ED(p, q)$. Especially, p (resp. q) is instead a binary vector for hamming distance.

2.3.2 Similarity-based Data Mining Algorithms

Similarity-based mining tasks often involve similarity computation among data objects. Such tasks include the algorithms of partitioning/density-based clustering, lazy learning-based classification, distance-based outlier detection [80]. Especially, kNN classification and k-means clustering are two of the most widely used mining algorithms [80, 95].

Given a query object q, kNN classification is to identify k objects nearest to q [80, 102]. kNN classification often deal with massive datasets at high dimensionality. For example, recent media search engines often transform an image to a high-dimensional vector of 50 to 1000+ dimensions [68]. To speed up the classification, index-based data structures such as hierarchical tree [29] show notable success on low-dimensional data, but suffer from "curse of dimensionality", degenerating to be linear scan when dimensionality is large [41, 143]. Distance bounds become an effective alternative for high-dimensional data. Unpromising candidates are pruned by the bounds, so as to reduce calling expensive exact computation. Several works [125, 177, 90, 214] proposed the lower bounds LB of exact Euclidean distance $ED(p, q)$ that satisfies $LB(p, q) \leqslant ED(p, q)$. The bounds often adopt dimensionality reduction techniques.

k-means clustering is an ubiquitous tool for data mining in many areas such as bioinformatics. For instance, gene expression data analysis adopts k-mean to identify the functionally related genes. It partitions genes into groups based on the similarity between their expression profiles [132]. *ED* is the most popular distance function for k-means [95]. Despite its age, Lloyd's algorithm has shown efficiency of converging in a small number of iterations to near-optimal solution [46]. Several studies [19, 78, 49] developed techniques to speedup the Lloyd's algorithm. The common idea is to use triangle inequality to reduce the distance computation between data points and centers.

2.4 Blockchain Mining

In this section, we first present the procedures of blockchain mining. We then discuss a common cryptographic hash algorithm, which is the basic computation task for blockchain mining.

2.4.1 Mining Process

Blockchain was introduced in 2008 as part of a proposal for a virtual currency system, bitcoin [144]. Its novel decentralized mechanism provides an effective solution to guarantee public privacy without threats to data integrity. Each party or participant has the right to access the data, but can not temper any historical data. In the past decade, Blockchain has been widely studied in database, IoT, network security, and digital financial [38].

The new data added into the blockchain can be any kind type of data, such as user trade transactions in banking systems [74]. Then the data are selected by participants in blockchain network who are trying to create new blocks. The participants received a lively name due to the impact of bitcoin - "*miner*". The process of creating new blocks is called "*mining*". The miners will verify the transactions to avoid any invalid transactions. Verifying data is not compulsory in blockchain, optional depending on applications. As blockchain is public, every participant has the right to create a new block. While there are many participants trying to create new blocks at same time, only one block will be accepted and added into the chain. Hence, to determine which block, blockchain needs *consensus mechanisms*, such as *proof-of-work* and *proof-of-stock* [76]. Especially, proof-of-work is the most common among blockchain platforms. Finally, after proof-of-work, the block will be broadcasted into the network and others will accept it.

Mining process is composed by two parts: Merkle tree construction and proof-of-

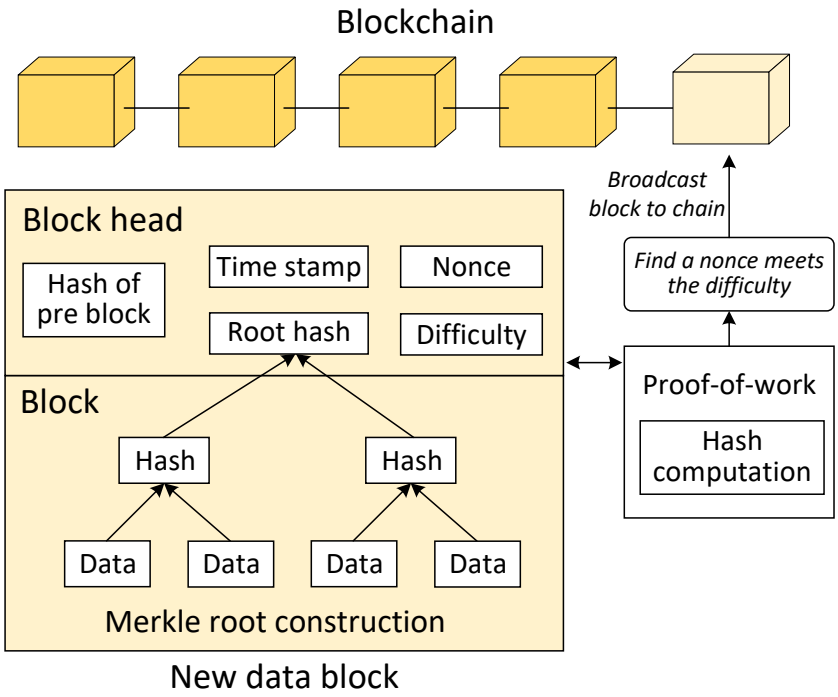


Figure 2.4: The overview process of blockchain mining.

work. Figure 2.4 illustrates the process of blockchain mining. Specifically, blockchain conducts the cryptographic hash computation on each new data. The hashes are stored in the manner of Merkle tree structure that each hash is linked to its parent following a parent-child tree-like relation. Eventually, there is one hash for the entire block at the tree root. Proof-of-work then solves a mathematical puzzle - finding a nonce value, which concatenates to the information at block header including the tree root hash to produce a hash that meets predefined difficult criteria. The criteria is generally a number *'difficulty'* starting with many zeros. Due to the high security of cryptographic hash computation, there is no more efficient way than blindly trying the random numbers until finding the nonce. Hence, the mining process requires massive computational efforts.

The expensive computation of blockchain mining leads to an obvious drawback: *performance*. New data in blockchain has high confirmation latency until physically

stored. In bitcoin [144], the latency is 10 minutes, and the throughput is 7 transactions per second [20]. In Ethereum [199], the throughput can only achieve up to 100s of transactions per second [47]. GPU is popular processor to accelerate the mining process due to its higher parallelism than CPU [141].

2.4.2 Cryptographic Hash Algorithm

Cryptographic hash function is the key component of blockchain mining to ensure data integrity, as discussed in Section 2.4.1. The mining process is to conduct massive hash computations. SHA is a family of cryptographic hash functions issued by the National Institute of Standards and Technology [31]. SHA-256 is a class of the SHA family, which generates a 256-bit message digest from data of an arbitrary size. SHA-256 is currently assumed as unbreakable and has been widely used in state-of-art blockchain systems and applications [40, 58].

2.5 Cardinality Estimation

Efficient cardinality estimation contributes to fast analytical query execution. In this section, we review the advances of cardinality estimation related to the proposed methods in this thesis. We first review the existing learning-based cardinality estimation, and then discuss the progressive cardinality estimation for query re-optimization.

2.5.1 Learning-based Cardinality Estimation

Given a query, cardinality estimation estimates the size of the intermediate result relations. The query optimizer relies on the estimated cardinalities to search the execution plans of high quality. The optimal execution plan can be identified if the estimations are error-free [122].

Histogram-based cardinality estimation methods [45, 5, 73] have been widely used in many industrial database systems as they are simple and have very low overhead. However, they cannot capture the data correlations among the tables as they make the attribute-value-independence assumption. *Sampling-based* approaches [34, 62, 188] outperform histogram-based methods as the correlations in data are naturally captured by data samples. However, sampling-based approaches have two limitations: (i) empty sampling set of join result; and (ii) high sampling overhead.

Recently, the database community recognized the potential of replacing traditional cardinality estimation methods by learning-based models (e.g., neural network, autoregressive model) [195, 118]. Existing learning-based estimators can be classified into three categories: *query-driven*, *data-driven* and *hybrid*.

Data-driven cardinality estimators [85, 212, 213] share the same idea with traditional cardinality estimation methods, i.e., they hope to capture the correlations and distributions of data across the tables. For example, [85] adopts relational sum product networks (RSPN) to capture the probability distribution among relations, and translates a query into the evaluations of probabilities and expectations based on RSPN. *Query-driven cardinality estimators* [172, 109, 175, 82, 53] formulate cardinality estimation as a regression problem. The contents of queries and their true cardinalities are used as training data to learn a mapping from queries to cardinalities. [52] uses regression techniques such as XGBoost, to train the model to produce approximate cardinality labels. *Hybrid cardinality estimator* [200] learns the joint data distribution among the tables as in the data-driven estimators, and uses query samples as auxiliary information at the same time.

2.5.2 Query Re-optimization

Though extensive efforts have been made to improve the accuracy of cardinality estimation, the errors can still be large for complex queries, e.g., those with multiple joins. Query re-optimization techniques [13, 99, 4, 51, 201, 146] have been proposed to combat the influence of large estimation errors on query optimization. There are two classes of query re-optimization techniques. (1) *Re-optimizing during query execution.* [4] samples data from the intermediate results, and tries different operator implementations and join orders on the samples to adjust the remaining execution plan. The re-optimization can be adopted into large-scale data processing systems such as Hadoop that continuously collecting statistics to feedback query optimizer during execution. (2) *Re-optimizing before query execution.* [100] proposes the "pilot runs" to execute the query over a set of data samples to estimate cardinalities, after which the optimizer relies on the cardinalities to choose an initial execution plan.

Chapter 3

Accelerating Similarity-based Mining Tasks by NVM PIM

Similarity computation is an essential building block in many mining tasks on multi-dimensional data. Examples of similarity-based mining tasks include kNN classification [33], k-means clustering [95], motif discovery and anomaly detection [142, 80]. These mining tasks often deal with massive datasets at high dimensionality. For instance, in kNN classification for images, each object is normally modeled as a vector with hundreds of dimensions, and every classification requires examining a database of millions of images.

The ever-growing gap between processor speed and memory speed brings about the *memory wall* issue. Contemporary processor speed improves at 70% annually but memory speed increases by only 7% annually [153]. This renders similarity computation in mining algorithms expensive due to the substantial amount of data transfer between processors and memory.

Recently, emerging NVM devices like ReRAM [9, 164], STT-RAM [97], and PCM [145], enable processing-in-memory [60], which is an efficient approach to diminish expensive data transfer (to processors) by directly processing the data stored in such devices. Among these candidates of NVM devices, ReRAM has superior characteristics such as lower read latency, higher data density and lower write en-

ergy, as discussed in Section 2.1. In the past decade, the industry (e.g., HP [174], Samsung [14], Toshiba [128]) has been developing ReRAM to support PIM.

In this chapter, we investigate *how to accelerate similarity-based mining tasks by using ReRAM PIM without compromising the accuracy of results.* Our challenges stem from the following characteristics of ReRAM. First, ReRAM relies on crossbars for processing data, and it only supports specific operations (e.g., dot-product or vector-matrix multiplication) but not arbitrary operations. Second, the operands in ReRAM PIM can only be non-negative integers (with limited precision), due to the nature of analog computation in ReRAM crossbars. Third, the write endurance of ReRAM is limited compared to DRAM, and the unessential data rewritten should be avoided.

We are aware of prior works on utilizing ReRAM PIM in several application domains, e.g., neural network [36, 164], graph computing [221, 81], and DNA alignment [88]. However, they have not studied how to accelerate similarity computation on multi-dimensional data.

To tackle the aforementioned challenges, we propose a framework that exploits ReRAM PIM to accelerate a given similarity-based data mining algorithm, while preserving the accuracy of results.

• First, we conduct performance profiling on similarity-based mining algorithms to identify their performance bottleneck, and then estimate the potential performance gain of using PIM.

• Second, we propose to decompose a similarity (or its bound) function into different parts so that: (i) most of the computation can be quickly performed by using ReRAM PIM, and (ii) the remaining parts can be executed in the host processor (e.g., CPU) with little data transfer. We establish PIM-aware bound computation so that the accuracy of results won't be compromised. In addition, we propose techniques for PIM memory management and execution plan optimization.

Experimental results on real datasets show that our proposed method achieves up to 11.0x and 8.5x speedup for state-of-art kNN classification and k-means clustering algorithms, respectively.

The remainder of this chapter is organized as follows. Section 3.1 introduces the background and motivation. Section 3.2 presents the overview of our framework. Section 3.3 and Section 3.4 discuss the details of our proposed framework. The experimental results are discussed in Section 3.5. Section 3.6 concludes the chapter with future research directions.

3.1 Preliminaries

In this section, we introduce the details of ReRAM PIM about how to support computation on multi-dimensional data at high-precision. We then review state-of-art similarity-based mining algorithms.

3.1.1 ReRAM Processing-in-memory

ReRAM PIM can support the dot-product efficient computation in analog manner. A ReRAM cell can switch its state among different resistance levels, which can be used to represent a 1-7 bits integer [10]. Each cell is connected by two orthogonal nanowires - wordline and bitline, and multiple wordlines and bitlines compose a *crossbar*. Figure 3.1 shows an example of dot-product computation on a 3x3 2-bit crossbar. Three 3-dimensional vectors [3, 1, 0], [1, 2, 3] and [2, 0, 1] as multipliers are pre-programmed vertically along bitlines. The multiplicand vector [3, 1, 2] is directly injected to wordlines after converting to input voltages. Then the crossbar concurrently generates three *dot-product* results.

Then we discuss how to deal with high-precision data ($b > h$), where b denotes the bit size of data operand, and h denotes the bit precision of a ReRAM cell. In this case, a b-bit operand (multiplier) is segmented into multiple h-bit parts and then

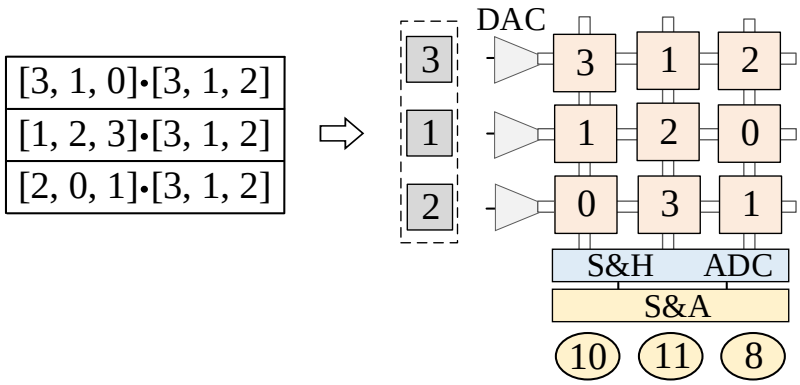


Figure 3.1: Example of PIM dot-product operation on ReRAM crossbar.

stored in adjacent cells of the same row. Similarly, multiplicands are segmented and converted to input voltages by every h bits. Dot-product is decomposed into several sub-operations, and the final result is obtained by *shifting* and *adding* result of each sub-operation with circuit S&A [169]. Figure 3.2 depicts an example of processing 6-bit data on 2-bit ReRAM cells. The decimal value '25', which is '011001' in binary, is segmented to '01' (1), '10' (2), '01' (1), and then stored in cells of the first row. For each sub-operation, the partial results are shifted and added to get final result.

Next we discuss how to compute dot-product on high-dimensional data, when the dimensionality exceeds the crossbar size. A crossbar is typically smaller than 1024×1024 [147]. Given a crossbar with size of $m \times m$, d-dimensional vector can be decomposed to a group of m-dimensional vectors, each of which is mapped to a crossbar (denoted as '*data crossbar*'). Then each data crossbar provides an output as partial results, and a crossbar (denoted as '*gather crossbar*') storing all-ones vector $\mathbf{e}=[1, ..., 1]$ is employed to sum up partial results vertically. Figure 3.3 presents an example of handling 6-dimensional vector on a 3x3 crossbar.

Due to the analog computation in crossbars, ReRAM PIM supports the dot-product operation on non-negative integer vectors only but not arbitrary operations. A few works proposed customized PIM designs for different applications.

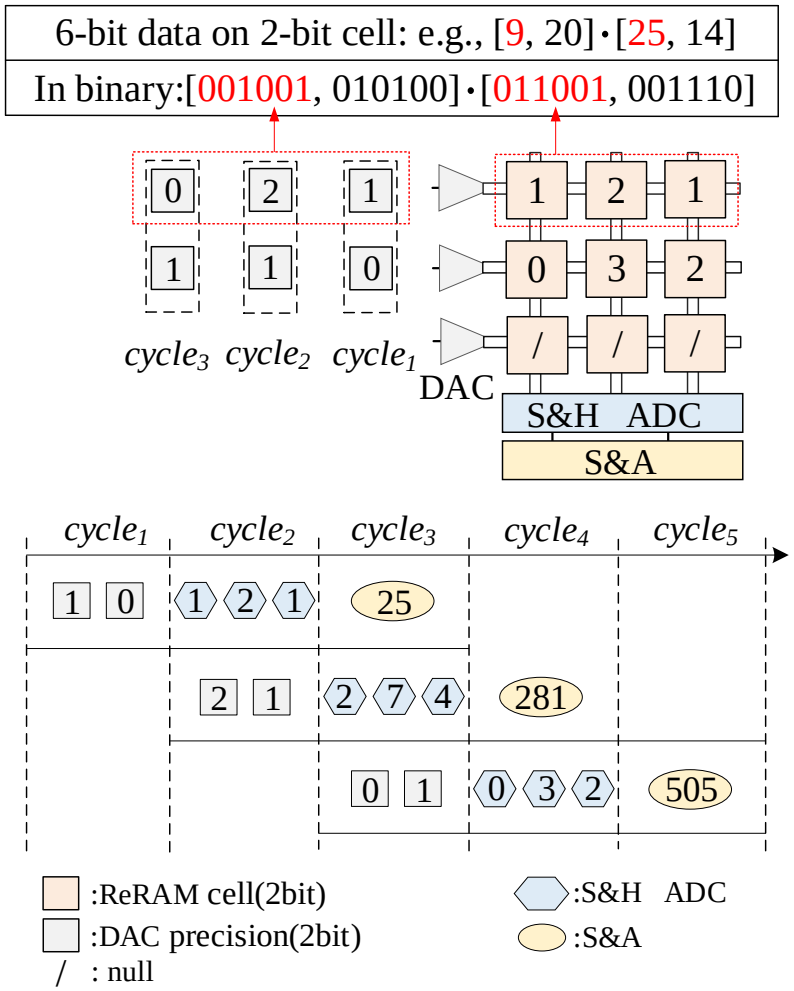


Figure 3.2: Example of pipeline of PIM dot-product operation on high-precision (i.e., $b>h$) data, example crossbar contains 3x3 2-bit cells. S&H is sample and hold circuit, S&A is shift and add circuit. DAC (ADC) is digital (analog)-to-analog (digital) converter.

ISAAC [164] adds sigmoid unit and ReLU logic to support the computing functionality in neural network; however, those units cannot be used in similarity computation. GraphR [169] uses fixed-point numbers to approximate floating-point values in graph computing, and claims that such precision loss is acceptable in the PageRank algorithm. However, such precision loss may compromise the accuracy of results in data mining tasks (e.g., kNN classification). In this work, we will utilize ReRAM PIM to compute bound functions and filter irrelevant objects, in order to preserve

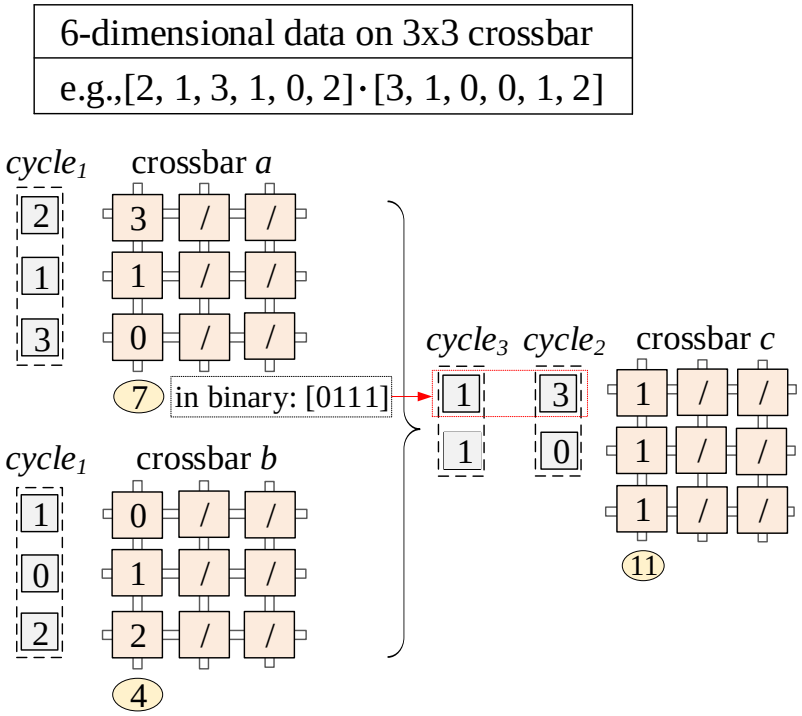


Figure 3.3: Example of PIM dot-product operation on high-dimensional (i.e., d>m) data, example crossbar contains 3x3 2-bit cells (omit peripheral circuits for simplicity).

the accuracy of results.

3.1.2 Similarity-based Data Mining Algorithms

We focus on the similarity-based mining tasks that involve similarity computation among data objects. Especially, kNN classification and k-means clustering are two of the most widely used mining algorithms [80, 95], and we use them as examples in this chapter. We introduce some relevant symbols in this chapter. Let p denote a d-dimensional vector in dataset D. p_i is the value on i-th dimension and takes b-bit. N is the number of vectors in D. q is another d-dimensional vector related to the applications, such as the query object in kNN classification. $dist(p, q)$ is distance between p and q, such as $ED(p, q)$.

kNN classification: While index structures such as hierarchical trees suffer from

"the curse of dimensionality", the distance bounds become an alternative to speed up the classification. Existing works [125, 214, 90, 177] proposed various bounds for different distance measures such as *ED*, *CS* and *PCC*. Table 3.1 shows representative bounds of *ED*, where prefix *LB* denotes lower bounds of exact distance $dist(p, q)$ that satisfies $LB(p, q) \leqslant dist(p, q)$. The bounds adopt dimensionality reduction techniques. For example, LB_{FNN} divides d-dimensional vector into d' segments with equal length l ($l \cdot d' = d$), and $\hat{p}_i$ denotes i-th segment. The bound is then computed by using mean value (e.g., $\mu(\hat{p}_i)$) and the standard deviation (e.g., $\sigma(\hat{p}_i)$) of each segment [90]. For *CS* and *PCC*, as terms independently involving p such as $\sqrt{\sum_{i=1}^{d} p_i^2}$, can be pre-computed, the computation can be reduced to the maximum dot-product search problem that retrieves vectors having large dot-product result with query vector. Prior works focus on devising upper bound *UB* that satisfies $UB(p, q) \geqslant dist(p, q)$, such as UB_{part} [177]. For kNN on *HD*, [138] observed that there is no obvious efficient technique significantly better than linear scan.

Table 3.1: Representative bounds for kNN classification: LB_{OST} [125], LB_{SM} [214], LB_{FNN} [90], and UB_{part} [177].

Symbol	Equation	Dis.
$LB_{OST}(p, q)$	$\sum_{i=1}^{d'}(p_i - q_i)^2 + (\sqrt{\sum_{i=d'+1}^{d} p_i^2} - \sqrt{\sum_{i=d'+1}^{d} q_i^2})^2$	*ED*
$LB_{SM}(p, q)$	$l \cdot \sum_{i=1}^{d'}(\mu(\hat{p}_i) - \mu(\hat{q}_i))^2$	*ED*
$LB_{FNN}(p, q)$	$l \cdot \sum_{i=1}^{d'}((\mu(\hat{p}_i) - \mu(\hat{q}_i))^2 + (\sigma(\hat{p}_i) - \sigma(\hat{q}_i))^2)$	*ED*
$UB_{part}(p, q)$	$\sum_{i=1}^{d'} p_i q_i + \sqrt{\sum_{i=d'+1}^{d} p_i^2}\sqrt{\sum_{i=d'+1}^{d} q_i^2}$	*PCC* *CS*

***k*-means clustering**: Lloyd's algorithm is the most common implementation for k-means clustering [46]. Two-step iterative refinement composes Lloyd's algorithm. In the assign step, each data point is assigned to the cluster whose center is the nearest. In the update step, centers are updated in accordance with assigned points. Several studies [56, 49, 46] developed techniques to speedup the Lloyd's algorithm,

which use triangle inequality to reduce distance computation between data points and centers. Let c be one of k cluster centers, and $c_a(p)$ be the center closest to data point p. Elkan [56] employs an upper and k lower bounds. The upper bound $UB(p)$ is bounding on the distance between p and $c_a(p)$ (i.e, $UB(p) \geqslant dist(p, c_a(p))$). For each center c, Elkan keeps a lower bound $LB(p, c)$ of distance between p and c. $UB(p)$ is updated by adding the moved distance of $c_a(p)$ after each iteration. The bounds are used to effectively avoid unnecessary *ED* distance calculation. For example, to determine whether c is closer to p, the calculation of $dist(p, c)$ can be avoided if $UB(p) \leqslant LB(p, c)$. Drake [49] and Yinyang [46] follow the similar strategy with employing less bounds.

3.2 Overview

In this section, we present a general ReRAM PIM architecture for speeding up similarity-based data mining algorithms, and briefly discuss the proposed systematic framework.

3.2.1 PIM Architecture

Figure 3.4 depicts an overview of heterogeneous architecture with ReRAM PIM, compared to conventional Neumann architecture. With ReRAM PIM, functionalities of storage and processing are integrated into the main memory. Host processor (e.g., CPU) supports complex arithmetic and logic computation but suffers from moving data through memory hierarchy. PIM provides efficient in-suit computation without moving data but supports specific operations. Host processor and PIM collaborate for general-purpose computation, yet are independently responsible for different tasks that meet respective superiority. In ReRAM-based memory, each bank contains three parts: *memory array*, *buffer array*, and *PIM array*.

Memory array works as standard memory for storage, playing the same role as

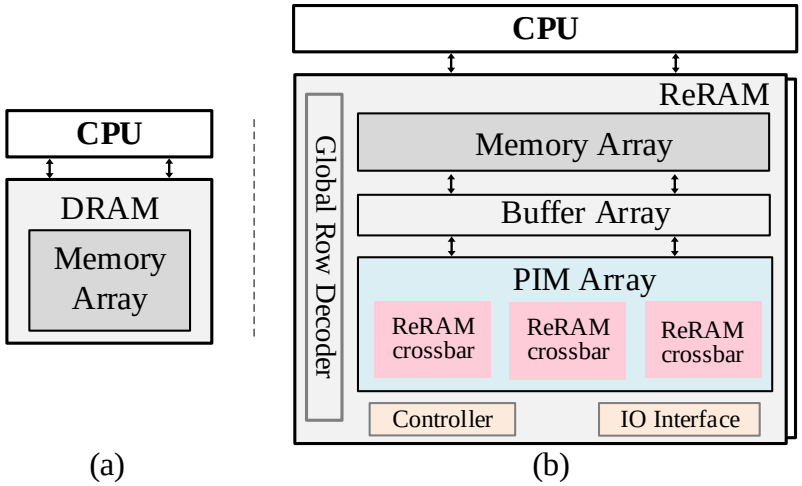


Figure 3.4: Overview of (a) conventional architecture and (b) ReRAM-based processing-in-memory architecture.

in traditional architecture to exchange data with the host processor. PIM array is composed of multiple crossbars that are capable of processing the resided data. Next to PIM array, there is buffer array used to cache PIM results, as the massive parallelism of PIM produces abundant results concurrently. With help of the buffer, PIM array can work with CPU in parallel. CPU can collect PIM results in buffer array without waiting for PIM array. Though PIM array heavily relies on ReRAM device, DRAM is flexible to be used for memory array to eliminate the impact of high access latency of ReRAM. Controller is instruction interface between software and hardware. It also coordinates the dataflow among different array parts. I/O interface is used to load data into memory array, or instructions into controller. Most data movement during PIM happens on memory internally and enjoys high in-memory bandwidth. This simple architecture is widely adopted in prior works [60, 140].

3.2.2 Systematic Framework

ReRAM PIM offers an opportunity of reducing data transfer from memory to processor. However, ReRAM PIM does not support arbitrary computation. Then two question arises.

- *Given a similarity-based mining algorithm, could ReRAM PIM be exploited to accelerate the computation?*
- *How to implement it efficiently without comprising the accuracy of results?*

We propose a framework to make a similarity-based mining algorithm aware of characteristics of ReRAM PIM. The limited functionality of PIM makes it infeasible to run the entire algorithm. We use PIM for the similarity computation, and host processor (e.g., CPU) to coordinate the remaining steps. Given an algorithm, the first step is to conduct *performance profiling* (Section 3.3), which helps to understand data transfer pressure of the algorithm and identify the function causing major bottleneck. If the function is a similarity or bound function feasible to expose most computational task as dot-product, we define it as *PIM-aware function* that can enjoy offloading computation to PIM (Section 3.4.1). We exploit PIM to assist processing the functions, benefiting from a significant decrease in data transfer. We next present *PIM-aware bound function* based on non-negative integers, which helps to prune unpromising candidates and guarantee correct result (Section 3.4.2). PIM array typically has limited space, which might be infeasible to accommodate the entire dataset. We then compress the dataset based on given hardware configuration to avoid re-programming the crossbars (Section 3.4.3). In addition, further optimization can be achieved by properly combining PIM and original operations in the algorithm (Section 3.4.4).

3.3 Algorithm Profiling

In this section, we present performance profiling which helps us identify the bottleneck of an algorithm, and estimate the potential performance gain of using PIM. The performance profiling can answer that whether ReRAM PIM could be exploited to speed up the algorithm. We present two kinds of profiling: 1) performance break-

down by hardware components; 2) performance breakdown by functions.

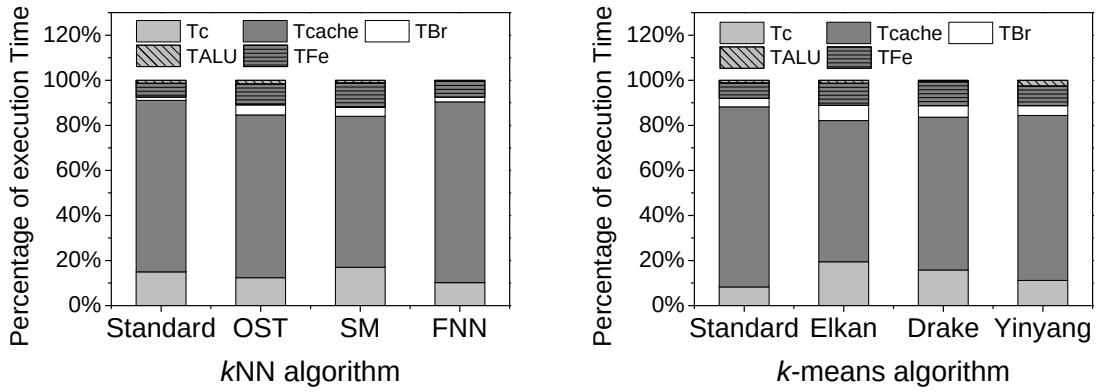
3.3.1 Performance Breakdown by Hardware Components

We first explain how to profile an algorithm in terms of hardware components. The profiling results can be used to justify whether it is promising to utilize PIM to accelerate the algorithm. We then show profiling results for the kNN and k-means algorithms. We conducted experiments on a machine with Intel Xeon E5-2620 CPU, 16 GB DRAM, and Linux 4.15.0. The algorithms are implemented in C++, compiled by GNU C++ with -O3 option, and executed in a single thread.

According to Intel performance analysis guide [124], the execution time of an algorithm on modern processors can be characterized as: 1) The computation time (T_c) is the actual time spent by CPU on computation. 2) The memory stall time (T_{cache}) is caused by L1/L2/L3 and TLB cache misses, which are triggered by data transfer from memory. 3) The ALU execution stall time (T_{ALU}) is caused by long latency operations such as division. 4) The branch misprediction stall time (T_{Br}) occurs because of branch misprediction in CPU. 5) The front-end stall time (T_{Fe}) occurs during fetching and decoding instructions. Total execution time (T_{total}) is as:

$$T_{total} = T_c + T_{cache} + T_{ALU} + T_{Br} + T_{Fe} \tag{3.1}$$

In modern processors, hardware counters are embedded to monitor processing states and provide statistics on performance metrics. PAPI [178, 2] is an operating system independent API to access the counters, which has been widely used for performance analysis. We used PAPI to measure time components in Equation 3.1. Note that the profiling by hardware components has the assumption of no major overlap of pipeline processing [7]. We adopt the profiling to roughly explore memory pressure of the algorithm, not compulsorily used in our proposed technical design. Such preliminary profiling can help us to easily identify the algorithm whether suffers



(a) MSD dataset, k=10 (b) NUS-WIDE dataset, k=64

Figure 3.5: Performance breakdown of representative kNN and k-means algorithms.

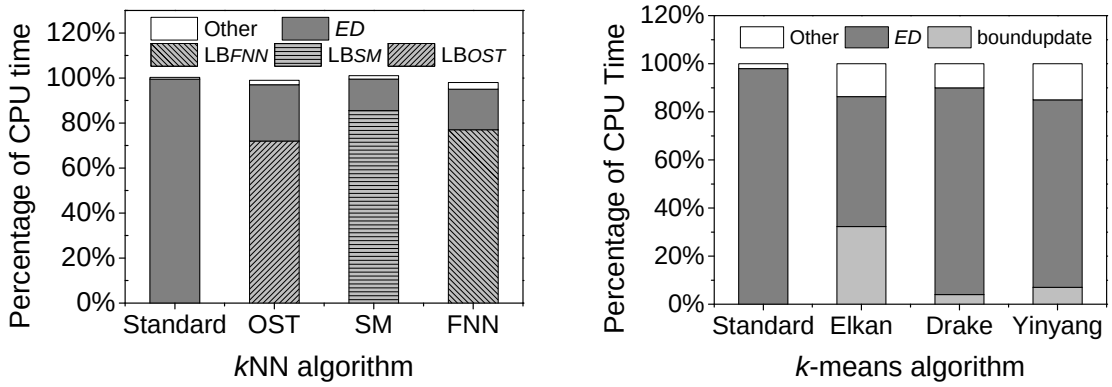
from data transfer, and thus save efforts to further employing PIM.

Table 3.4 (in Section 3.5) lists the statistics of datasets for experiments. We fix k as 10 for kNN. We show profiling results of state-of-art algorithms FNN [90], OST [125], and SM [214], which employ the bounds on *ED* discussed in Section 3.1.2. For empirical completeness, we also show standard linear scan method ('Standard'). For k-means, we conducted profiling on methods discussed in Section 3.1.2 including standard Lloyd's algorithm ('Standard'). The same initial 64 (k=64) centers are chosen.

Figure 3.5 depicts the proportion of each component on total execution time. For kNN algorithms, we observe that T_{cache} dominates the stall time, accounting for 65-83% of total time. For k-means algorithms, the majority (62-75%) of execution time is caused by T_{cache}. The profiling results imply that the kNN and k-means algorithms suffer from the latency of data transfer. It is thus promising to exploit PIM for optimization.

3.3.2 Performance Breakdown by Functions

We then discuss the profiling of an algorithm by functions, in order to identify the function causing performance bottleneck. The execution time of an algorithm can be



(a) MSD dataset, k=10 (b) NUS-WIDE dataset, k=64

Figure 3.6: Execution time breakdown of representative kNN and k-means algorithms.

decomposed to the components spent on each function and time T_{other} caused by all other operations such as condition check. Assume the algorithm contains t functions f_1, f_2, ..., f_t, and the time spent on each function is T_{f_1}, T_{f_2}, ..., T_{f_t} respectively. Then total execution time T_{total} of algorithm is T_{total}=$\sum_{i=1}^{t} T_{f_i} + T_{other}$. To measure each T_{f_i}, we can simply adopt time system calls such as $clock()$ and $clock\_gettime()$ that provide fine-grain measurement at function-level.

Figures 3.6 depicts the execution time breakdown of the kNN and k-means algorithms. For kNN algorithms, calculation of *ED* dominates the total execution time for Standard, and bound functions such as LB_{FNN} incur the majority (72%-86%) of total time for other algorithms. For k-means algorithms, calculation of *ED* takes 52-96% of the execution time. The profiling results indicate that the distance computation causes the performance bottleneck of the algorithm, which is expected to be optimized.

3.3.3 Potential Performance Gain of using PIM

The profiling results in Section 3.3.2 also help us estimate the optimal performance gain of using PIM. Recall that the execution time of an algorithm can be character-

ized using time spent on each function. Let $\mathbb{F}$ be the subset of functions that can be optimized with PIM (we shall discuss the functions in set $\mathbb{F}$ in Section 3.4). Ideally, if the computation time of functions in $\mathbb{F}$ can be reduced to 0, we have the theoretical optimal running time $T_{PIM-oracle}$ as:

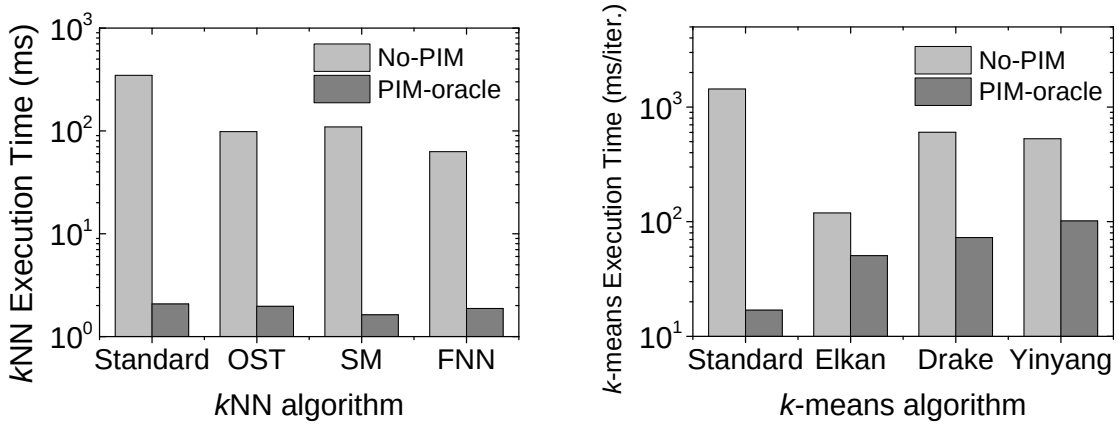
$$T_{PIM-oracle} = T_{total} - \sum_{f_i \in \mathbb{F}} T_{f_i} \tag{3.2}$$

Note that $T_{PIM-oracle}$ serves as a lower bound of any PIM implementation of the given algorithm. This insight of $T_{PIM-oracle}$ helps us further verify the adoption of PIM. If $T_{PIM-oracle}$ is high and close to T_{total}, we might not consider exploiting PIM.

Figure 3.7 plots the execution time of No-PIM and PIM-oracle for kNN and k-means, where No-PIM is the native algorithm running on CPU, and PIM-oracle is based on the estimation of Equation 3.2. Figure 3.7(a) shows that PIM-oracle is much faster than No-PIM for the kNN algorithms. For instance, PIM-oracle is 183.9x faster than No-PIM for Standard. The high potential improvement is because that the function set $\mathbb{F}$ includes *ED* and bound functions such as LB_{FNN}, which accounts for the most computation overhead. This implies that it is promising to speed up the kNN algorithms with PIM. Besides, for k-means algorithms, the set $\mathbb{F}$ contains distance *ED* (in assign step). For Standard, PIM-oracle is 51.4x faster than No-PIM. However, PIM-oracle is 7.5x, 5.3x and 2.2x faster than No-PIM respectively, for Drake, Yinyang and Elkan. The lower potential improvement is because the distance computation accounts for comparatively less overhead of the algorithms.

3.4 Accelerating Algorithm With PIM

In this section. we present how to compute the similarity and bound functions with ReRAM PIM by addressing the limitations. In addition, we propose an optimization to further reduce the data transfer cost of algorithm.



(a) MSD dataset, k=10 (b) NUS-WIDE dataset, k=64

Figure 3.7: Performance comparison between algorithms without PIM and PIM-oracle (i.e., $T_{PIM-oracle}$) for kNN and k-means.

3.4.1 PIM-aware Function Decomposition

We identify a similarity or bound function as *PIM-aware function* if it can be decomposed to expose most computation as dot-product. Recall that ReRAM crossbars support specific operations (e.g., dot-product) but not arbitrary operations. Fortunately, it is possible to decompose a function into two parts: (i) sub-operations that can be processed by ReRAM PIM, and (ii) sub-operations that can be pre-computed. PIM readily processes on vector data at the online stage, the results of which are merged with pre-computed data in host processor for the final result. ReRAM PIM and host processor compute the similarity or bound function in the manner of co-processing. In this way, we decompose similarity or bound function $F(p, q)$ into PIM-aware format as follows:

$$F(p, q) = G(\Phi(p), \Phi(q), p \cdot q) \tag{3.3}$$

• $\Phi(p)$ takes a vector p from a dataset D as input and returns a fixed-size output. This function can be computed at offline stage. The same function $\Phi(q)$ can be applied on a given vector q involving applications, such as query object in kNN. It

Table 3.2: PIM-aware decomposition of similarity function and bound function.

	Offline	Online		
Function	$\Phi(p)$	$\Phi(q)$	$p \cdot q$	G
ED	$\sum_{i=1}^{d} p_i^2$	$\sum_{i=1}^{d} q_i^2$	$\sum_{i=1}^{d} p_i q_i$	$\Phi(p) + \Phi(q) - 2 \cdot p \cdot q$
CS	$\sqrt{\sum_{i=1}^{d} p_i^2}$	$\sqrt{\sum_{i=1}^{d} q_i^2}$	$\sum_{i=1}^{d} p_i q_i$	$\frac{p \cdot q}{\Phi(p)\Phi(q)}$
PCC	$\frac{\Phi_a: \sqrt{d \sum_{i=1}^{d} p_i^2 - (\sum_{i=1}^{d} p_i)^2}}{\Phi_b: \sum_{i=1}^{d} p_i}$	$\frac{\Phi_a: \sqrt{d \sum_{i=1}^{d} q_i^2 - (\sum_{i=1}^{d} q_i)^2}}{\Phi_b: \sum_{i=1}^{d} q_i}$	$\sum_{i=1}^{d} p_i q_i$	$\frac{d \cdot p \cdot q - \Phi_b(p)\Phi_b(q)}{\Phi_a(p)\Phi_a(q)}$
HD	$\tilde{p_i} = \begin{cases} 0 & \text{if } p_i = 1 \\ 1 & \text{if } p_i = 0 \end{cases}$	$\tilde{q_i} = \begin{cases} 0 & \text{if } q_i = 1 \\ 1 & \text{if } q_i = 0 \end{cases}$	$\frac{p \cdot q: \sum_{i=1}^{d} p_i q_i}{\tilde{p} \cdot \tilde{q}: \sum_{i=1}^{d} \tilde{p_i} \tilde{q_i}}$	$d - p \cdot q - \tilde{p} \cdot \tilde{q}$
LB_{FNN}	$l \cdot \sum_{i=1}^{d'} (\mu(\hat{p_i})^2 + \sigma(\hat{p_i})^2)$	$l \cdot \sum_{i=1}^{d'} (\mu(\hat{q_i})^2 + \sigma(\hat{q_i})^2)$	$\frac{\mu(\hat{p}) \cdot \mu(\hat{q}): \sum_{i=1}^{d'} \mu(\hat{p_i})\mu(\hat{q_i})}{\sigma(\hat{p}) \cdot \sigma(\hat{q}): \sum_{i=1}^{d'} \sigma(\hat{p_i})\sigma(\hat{q_i})}$	$\Phi(p)+\Phi(q)-2l \cdot \mu(\hat{p}) \cdot \mu(\hat{q})$ $-2l \cdot \sigma(\hat{p}) \cdot \sigma(\hat{q})$
LB_{SM}	$l \cdot \sum_{i=1}^{d'} \mu(\hat{p_i})^2$	$l \cdot \sum_{i=1}^{d'} \mu(\hat{q_i})^2$	$\sum_{i=1}^{d'} \mu(\hat{p_i})\mu(\hat{q_i})$	$\Phi(p) + \Phi(q) - 2l \cdot p \cdot q$
LB_{OST}	$\frac{\Phi_a: \sum_{i=1}^{d'} p_i^2}{\Phi_b: \sqrt{\sum_{i=d'}^{d+1} p_i^2}}$	$\frac{\Phi_a: \sum_{i=1}^{d'} q_i^2}{\Phi_b: \sqrt{\sum_{i=d'}^{d+1} q_i^2}}$	$\sum_{i=1}^{d'} p_i q_i$	$\Phi_a(p) + \Phi_a(q) - 2 \cdot p \cdot q$ $+(\Phi_b(p) - \Phi_b(q))^2$
UB_{part}	$\sqrt{\sum_{i=d'+1}^{d} p_i^2}$	$\sqrt{\sum_{i=d'+1}^{d} q_i^2}$	$\sum_{i=1}^{d'} p_i q_i$	$\Phi(p) \cdot \Phi(q) + p \cdot q$

suffices to evaluate $\Phi(q)$ once. This can be computed in the host processor at the online stage.

• The dot-product operation $p \cdot q$ can be computed on PIM. This requires only constant data transfer cost.

• The function G is used to combine $\Phi(p)$, $\Phi(q)$ and $p \cdot q$ into final result of $F(p, q)$ at online stage. This function can be calculated in host processor because of the constant time complexity.

ED is a PIM-aware function as it can be rewritten as follows:

$$\underbrace{ED(p,q)}_{F} = \underbrace{\sum_{i=1}^{d} p_i^2}_{\Phi(p)} + \underbrace{\sum_{i=1}^{d} q_i^2}_{\Phi(q)} - 2 \underbrace{\sum_{i=1}^{d} p_i \cdot q_i}_{p \cdot q} \tag{3.4}$$

Indeed, similarity function *CS*, *PCC* and *HD*, and bound functions in Table 3.1 are also PIM-aware functions. We list the PIM-aware function of each similarity or bound function in Table 3.2.

PIM-aware function can enjoy offloading most computation to PIM. The benefit is significant reduction of data transfer. As depicted in Figure 3.8, computing $ED(p, q)$ demands transferring $d \cdot b$ bits between memory and CPU on conventional

architectures, where b is bit size of each operand. In contrast, computation on vector data in G is done by pre-processing or PIM, which reduces the data transfer to $3{\cdot}b$ bits. Exploiting PIM requires the vector data from dataset are pre-programmed on crossbars in PIM array. For example, vector p for computing ED costs the crossbar space of $N \cdot d \cdot b$ bits. PIM array might have insufficient capacity to maintain them. Section 3.4.3 discusses how to compress the dataset based on given hardware capacity.

3.4.2 PIM-aware Bound Computation

Some similarity functions such as HD typically operate on integer vectors, so they can be computed exactly by using PIM. The other similarity or bound functions such as ED take often floating-point vectors, so PIM cannot be used to directly compute the exact value of the functions. Hence, the correct results of algorithm cannot be guaranteed.

ReRAM PIM demands specific data type (i.e., non-negative integer), and it cannot readily support floating-point values. To tackle this, we utilize PIM to compute lower/upper bounds of distance functions, which still benefits from the significant reduction on data transfer. The lower/upper bounds are used to prune unpromising objects following filter-and-refinement strategy, which still guarantees the correct results. For example, while the update step of k-means algorithms requires exact

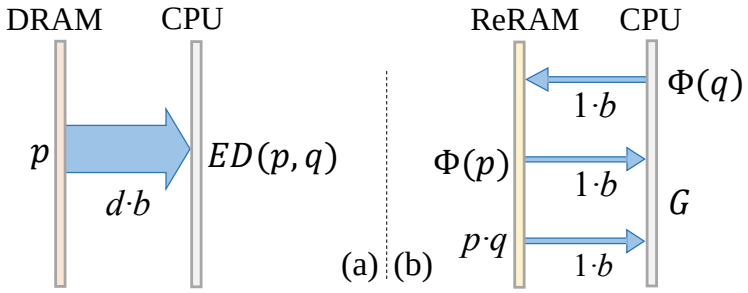


Figure 3.8: Overview of data transfer cost of computing (a) $ED(p;q)$ and (b) G (unit: Bit).

computation of *ED*, PIM can accelerate the algorithms by supplying lower bounds of exact *ED* in assign step. The far-away centers are readily pruned by invoking comparison between bounding values and the threshold (i.e., the distance to currently assigned center).

Given dataset D, we initially normalize the floating-point values to be range of [0, 1]. Scalar p_i is non-negative value within [0, 1]. We then enlarge p_i by multiplying constant α as scaling factor, and truncate the integer part $\lfloor \bar{p_i} \rfloor$. Then we have a vector $\lfloor \bar{p} \rfloor$ with only non-negative integers:

$$\bar{p_i} = p_i \cdot \alpha \tag{3.5}$$

$$\lfloor \bar{p} \rfloor = (\lfloor \bar{p_1} \rfloor, \lfloor \bar{p_2} \rfloor, ..., \lfloor \bar{p_d} \rfloor) \tag{3.6}$$

Similarly, we can have non-negative integer vector $\lfloor \bar{q} \rfloor$. We then propose *PIM-aware bound function*, which serves as a bound of PIM-aware function. The involved dot-product operation just deals with non-negative integer vectors. As discussed in Section 3.4.1, *ED*, *CS*, *PCC*, and their bound functions (in Table 3.1) are PIM-aware functions. For *ED* and its bounds such as LB_{FNN}, we are interested in their lower bounds. For *CS*, *PCC*, and the related bounds, we are interested in their upper bounds. We then present the PIM-aware bound of the similarity and bound functions as follows.

Theorem 1. *Squared Euclidean distance of two d-dimensional vectors p and q has a lower bound:*

$$LB_{PIM-ED}(p,q) = \frac{1}{\alpha^2}(\Phi(\bar{p}) + \Phi(\bar{q}) - 2 \cdot \lfloor \bar{p} \rfloor \cdot \lfloor \bar{q} \rfloor - 2d) \tag{3.7}$$

where $\Phi(\bar{p}) = \sum_{i=1}^{d} \bar{p_i}^2 - 2\sum_{i=1}^{d} \lfloor \bar{p_i} \rfloor$ *(resp.* $\Phi(\bar{q})$*), and* $\lfloor \bar{p} \rfloor \cdot \lfloor \bar{q} \rfloor = \sum_{i=1}^{d} \lfloor \bar{p_i} \rfloor \lfloor \bar{q_i} \rfloor$*. Here, d is number of dimensions, and* $\bar{p_i}$ *is the positive floating-point value normalized from* p_i *with* α*,* $\lfloor \bar{p_i} \rfloor$ *is integer part of* $\bar{p_i}$*.* $\Phi(\bar{p})$ *(*$\Phi(\bar{q})$*) is a floating-point value.*

Proof. As $\lfloor \bar{p}_i \rfloor$ ($\lfloor \bar{q}_i \rfloor$) is integer part of $\bar{p}_i$ ($\bar{q}_i$), we have:

$$\lfloor \bar{p}_i \rfloor \lfloor \bar{q}_i \rfloor \leqslant \bar{p}_i \bar{q}_i \leqslant (\lfloor \bar{p}_i \rfloor + 1)(\lfloor \bar{q}_i \rfloor + 1)$$

then we have lower bound:

$$\begin{aligned} LB_{PIM-ED}(p,q) &= \frac{1}{\alpha^2}(\Phi(\bar{p}) + \Phi(\bar{q}) - 2 \cdot \lfloor \bar{p} \rfloor \cdot \lfloor \bar{q} \rfloor - 2d) \\ &= \frac{1}{\alpha^2} \sum_{i=1}^{d} (\bar{p_i}^2 + \bar{q_i}^2 - 2\lfloor \bar{p_i} \rfloor \lfloor \bar{q_i} \rfloor - 2\lfloor \bar{p_i} \rfloor - 2\lfloor \bar{q_i} \rfloor - 2) \\ &= \frac{1}{\alpha^2} (\sum_{i=1}^{d} (\bar{p_i}^2 + \bar{q_i}^2) - 2 \sum_{i=1}^{d} (\lfloor \bar{p_i} \rfloor + 1)(\lfloor \bar{q_i} \rfloor + 1)) \\ &\leqslant \frac{1}{\alpha^2} \sum_{i=1}^{d} (\bar{p_i}^2 + \bar{q_i}^2 - 2\bar{p_i}\bar{q_i}) = ED(p,q) \qquad \square \end{aligned}$$

Theorem 2. *Cosine similarity between two d-dimensional vectors p, q has an upper bound as following:*

$$UB_{PIM-CS}(p,q) = \frac{\lfloor \bar{p} \rfloor \cdot \lfloor \bar{q} \rfloor + \Phi_b(\lfloor \bar{p} \rfloor) + \Phi_b(\lfloor \bar{q} \rfloor) + d}{\Phi_a(\bar{p}) \Phi_a(\bar{q})} \tag{3.8}$$

where $\Phi_a(\bar{p}) = \sqrt{\sum_{i=1}^{d} \bar{p_i}^2}$, $\Phi_b(\lfloor \bar{p} \rfloor) = \sum_{i=1}^{d} \lfloor \bar{p_i} \rfloor$ *(resp.* $\Phi_a(\bar{q})$ *and* $\Phi_b(\bar{q})$*), and* $\lfloor \bar{p} \rfloor \cdot \lfloor \bar{q} \rfloor = \sum_{i=1}^{d} \lfloor \bar{p_i} \rfloor \lfloor \bar{q_i} \rfloor$*. Here, d is number of dimensions, and* $\bar{p_i}$ *is the positive floating-point value normalized from* p_i *with* α*,* $\lfloor \bar{p_i} \rfloor$ *is integer part of* $\bar{p_i}$*.* $\Phi_a(\bar{p})$ *(*$\Phi_a(\bar{q})$*) and* $\Phi_b(\lfloor \bar{p} \rfloor)$ *(*$\Phi_b(\lfloor \bar{q} \rfloor)$*) are floating-point values.*

Proof. As $\lfloor \bar{p_i} \rfloor$ ($\lfloor \bar{q_i} \rfloor$) is integer part of $\bar{p_i}$ ($\bar{q_i}$), we have:

$$\bar{p_i}\bar{q_i} \leqslant (\lfloor \bar{p_i} \rfloor + 1)(\lfloor \bar{q_i} \rfloor + 1)$$

then we have upper bound:

$$
\begin{aligned}
UB_{PIM-CS}(p,q) &= \frac{\sum_{i=1}^{d}(\lfloor\bar{p}_i\rfloor\lfloor\bar{q}_i\rfloor + \lfloor\bar{p}_i\rfloor + \lfloor\bar{q}_i\rfloor + 1)}{\sqrt{\sum_{i=1}^{d}\bar{p}_i{}^2}\sqrt{\sum_{i=1}^{d}\bar{q}_i{}^2}} \\
&= \frac{\sum_{i=1}^{d}((\lfloor\bar{p}_i\rfloor + 1)(\lfloor\bar{q}_i\rfloor + 1))}{\sqrt{\sum_{i=1}^{d}\bar{p}_i{}^2}\sqrt{\sum_{i=1}^{d}\bar{q}_i{}^2}} \\
&\geqslant \frac{\sum_{i=1}^{d}\bar{p}_i\bar{q}_i}{\sqrt{\sum_{i=1}^{d}\bar{p}_i{}^2}\sqrt{\sum_{i=1}^{d}\bar{q}_i{}^2}} \\
&\geqslant \frac{\alpha^2\sum_{i=1}^{d}p_i\cdot q_i}{\alpha^2\sqrt{\sum_{i=1}^{d}p_i^2}\sqrt{\sum_{i=1}^{d}q_i^2}} = CS(p,q)
\end{aligned}
$$

□

Theorem 3. *Pearson correlation coefficient between two d-dimensional vectors p, q has an upper bound as following:*

$$
UB_{PIM-PCC}(p,q) = \frac{d\cdot\lfloor\bar{p}\rfloor\cdot\lfloor\bar{q}\rfloor - \Phi_b(\bar{p})\Phi_b(\bar{q}) + d\Phi_c(\lfloor\bar{p}\rfloor) + d\Phi_c(\lfloor\bar{q}\rfloor) + d^2}{\Phi_a(\bar{p})\Phi_a(\bar{q})} \tag{3.9}
$$

where $\Phi_a(\bar{p}) = \sqrt{d\sum_{i=1}^{d}\bar{p}_i{}^2 - (\sum_{i=1}^{d}\bar{p}_i)^2}$, $\Phi_b(\bar{p}) = \sum_{i=1}^{d}\bar{p}_i$, *and* $\Phi_c(\lfloor\bar{p}\rfloor) = \sum_{i=1}^{d}\lfloor\bar{p}_i\rfloor$, *(resp.* $\Phi_a(\bar{q})$, $\Phi_b(\bar{q})$ *and* $\Phi_c(\lfloor\bar{q}\rfloor)$*), and* $\lfloor\bar{p}\rfloor\cdot\lfloor\bar{q}\rfloor = \sum_{i=1}^{d}\lfloor\bar{p}_i\rfloor\lfloor\bar{q}_i\rfloor$. *Here, d is number of dimensions, and* $\bar{p}_i$ *is the positive floating-point value normalized from* p_i *with* α, $\lfloor\bar{p}_i\rfloor$ *is integer part of* $\bar{p}_i$. $\Phi_a(\bar{p})$ *(*$\Phi_a(\bar{q})$*),* $\Phi_b(\bar{p})$ *(*$\Phi_b(\bar{q})$*) and* $\Phi_c(\lfloor\bar{p}\rfloor)$ *(*$\Phi_c\lfloor\bar{q}\rfloor$*) are floating-point values.*

Proof.

$$
\begin{aligned}
UB_{PIM\text{–}PCC}(p,q) &= \frac{d\sum_{i=1}^{d}\lfloor\bar{p}_i\rfloor\lfloor\bar{q}_i\rfloor - \sum_{i=1}^{d}\bar{p}_i\sum_{i=1}^{d}\bar{q}_i + d\sum_{i=1}^{d}\lfloor\bar{p}_i\rfloor + d\sum_{i=1}^{d}\lfloor\bar{q}_i\rfloor + d^2}{\sqrt{d\sum_{i=1}^{d}\bar{p}_i{}^2 - (\sum_{i=1}^{d}\bar{p}_i)^2}\sqrt{d\sum_{i=1}^{d}\bar{q}_i{}^2 - (\sum_{i=1}^{d}\bar{q}_i)^2}} \\
&= \frac{d\sum_{i=1}^{d}((\lfloor\bar{p}_i\rfloor+1)(\lfloor\bar{q}_i\rfloor+1)) - \sum_{i=1}^{d}\bar{p}_i\sum_{i=1}^{d}\bar{q}_i}{\sqrt{d\sum_{i=1}^{d}\bar{p}_i{}^2 - (\sum_{i=1}^{d}\bar{p}_i)^2}\sqrt{d\sum_{i=1}^{d}\bar{q}_i{}^2 - (\sum_{i=1}^{d}\bar{q}_i)^2}} \\
&\geqslant \frac{d\sum_{i=1}^{d}\bar{p}_i\bar{q}_i - \sum_{i=1}^{d}\bar{p}_i\sum_{i=1}^{d}\bar{q}_i}{\sqrt{d\sum_{i=1}^{d}\bar{p}_i{}^2 - (\sum_{i=1}^{d}\bar{p}_i)^2}\sqrt{d\sum_{i=1}^{d}\bar{q}_i{}^2 - (\sum_{i=1}^{d}\bar{q}_i)^2}} \\
&\geqslant \frac{\alpha^2(d\sum_{i=1}^{d}p_iq_i - \sum_{i=1}^{d}p_i\sum_{i=1}^{d}q_i)}{\alpha^2\sqrt{d\sum_{i=1}^{d}p_i^2 - (\sum_{i=1}^{d}p_i)^2}\sqrt{d\sum_{i=1}^{d}q_i^2 - (\sum_{i=1}^{d}q_i)^2}} = PCC(p,q)
\end{aligned}
$$

□

Theorem 4. *Squared Euclidean distance between two d-dimensional p and q has a lower bound:*

$$
LB_{PIM\text{–}FNN}(p,q) = \frac{l}{\alpha^2}(\Phi(\hat{p}) + \Phi(\hat{q}) - 2\cdot\lfloor\mu(\hat{p})\rfloor\cdot\lfloor\mu(\hat{q})\rfloor - 2\cdot\lfloor\sigma(\hat{p})\rfloor\cdot\lfloor\sigma(\hat{q})\rfloor - 4d') \tag{3.10}
$$

where $\Phi(\hat{p}) = \sum_{i=1}^{d'}\mu(\hat{p}_i)^2 + \sum_{i=1}^{d'}\sigma(\hat{p}_i)^2 - 2\sum_{i=1}^{d'}\lfloor\mu(\hat{p}_i)\rfloor - 2\sum_{i=1}^{d'}\lfloor\sigma(\hat{p}_i)\rfloor$ *(resp.*$\Phi(\hat{q})$*).* $\lfloor\mu(\hat{p})\rfloor\cdot\lfloor\mu(\hat{q})\rfloor = \sum_{i=1}^{d'}\lfloor\mu(\hat{p}_i)\rfloor\lfloor\mu(\hat{q}_i)\rfloor$*, and* $\lfloor\sigma(\hat{p})\rfloor\cdot\lfloor\sigma(\hat{q})\rfloor = \sum_{i=1}^{d'}\lfloor\sigma(\hat{p}_i)\rfloor\lfloor\sigma(\hat{q}_i)\rfloor$*. Here, d-dimensional* $\bar{p}$ *is partitioned into* d' *segments of same length and* $\hat{p}_i$ *is i-th segment.* $\mu(\hat{p}_i)$ *and* $\sigma(\hat{p}_i)$ *are mean and standard deviation of i-th segment,* $\lfloor\mu(\hat{p}_i)\rfloor$ *and* $\lfloor\sigma(\hat{p}_i)\rfloor$ *are their integer parts.* $\Phi(\hat{p})$ *(*$\Phi(\hat{q})$*) is a floating-point value.*

Proof. As $\lfloor\hat{p}_i\rfloor$ ($\lfloor\hat{q}_i\rfloor$) is integer part of $\hat{p}_i$ ($\hat{q}_i$), we have:

$$
\lfloor\hat{p}_i\rfloor\lfloor\hat{q}_i\rfloor \leqslant \hat{p}_i\hat{q}_i \leqslant (\lfloor\hat{p}_i\rfloor+1)(\lfloor\hat{q}_i\rfloor+1)
$$

then we have lower bound:

$$
\begin{aligned}
LB_{PIM\text{–}FNN}(p,q) &= \frac{l}{\alpha^2}\sum_{i=1}^{d'}(\mu(\hat{p_i})^2+\mu(\hat{q_i})^2-2\lfloor\mu(\hat{p_i})\rfloor-2\lfloor\mu(\hat{q_i})\rfloor-2\lfloor\mu(\hat{p_i})\rfloor\lfloor\mu(\hat{q_i})\rfloor \\
&\qquad +\sigma(\hat{p_i})^2+\sigma(\hat{q_i})^2-2\lfloor\sigma(\hat{p_i})\rfloor-2\lfloor\sigma(\hat{q_i})\rfloor-2\lfloor\sigma(\hat{p_i})\rfloor\lfloor\sigma(\hat{q_i})\rfloor-4) \\
&= \frac{l}{\alpha^2}\sum_{i=1}^{d'}(\mu(\hat{p_i})^2+\mu(\hat{q_i})^2-2(\lfloor\mu(\hat{p_i})\rfloor+1)(\lfloor\mu(\hat{q_i})\rfloor+1) \\
&\qquad +\sigma(\hat{p_i})^2+\sigma(\hat{q_i})^2-2(\lfloor\sigma(\hat{p_i})\rfloor+1)(\lfloor\sigma(\hat{q_i})\rfloor+1)) \\
&\leqslant \frac{l}{\alpha^2}\sum_{i=1}^{d'}(\mu(\hat{p_i})^2+\mu(\hat{q_i})^2-2\mu(\hat{p_i})\mu(\hat{q_i})+\sigma(\hat{p_i})^2+\sigma(\hat{q_i})^2-2\sigma(\hat{p_i})\sigma(\hat{q_i})) \\
&\leqslant \frac{l}{\alpha^2}\sum_{i=1}^{d'}((\mu(\hat{p_i})-\mu(\hat{q_i}))^2+(\sigma(\hat{p_i})-\sigma(\hat{q_i}))^2) = LB_{FNN}(p,q) \\
&\leqslant ED(p,q)
\end{aligned}
$$

□

Theorem 5. *Squared Euclidean distance between two d-dimensional p and q has a lower bound:*

$$
LB_{PIM\text{–}SM}(p,q) = \frac{l}{\alpha^2}(\Phi(\hat{p})+\Phi(\hat{q})-2\cdot\lfloor\mu(\hat{p})\rfloor\cdot\lfloor\mu(\hat{q})\rfloor-2d') \tag{3.11}
$$

where $\Phi(\hat{p})=\sum_{i=1}^{d'}\mu(\hat{p_i})^2-2\sum_{i=1}^{d'}\lfloor\mu(\hat{p_i})\rfloor$ *(resp.*$\Phi(\hat{q})$*), and* $\lfloor\mu(\hat{p})\rfloor\cdot\lfloor\mu(\hat{q})\rfloor=\sum_{i=1}^{d'}\lfloor\mu(\hat{p_i})\rfloor\lfloor\mu(\hat{q_i})\rfloor$*. Here, d-dimensional* $\bar{p}$ *is partitioned into* d' *segments of same length.* $\hat{p_i}$ *is i-th segment.* $\mu(\hat{p_i})$ *is mean of i-th segment, and* $\lfloor\mu(\hat{p_i})\rfloor$ *is the integer part.* $\Phi(\hat{p})$ *(*$\Phi(\hat{q})$*) is a floating-point value.*

Proof.

$$
\begin{aligned}
LB_{PIM\text{–}SM}(p,q) &= \frac{l}{\alpha^2}\sum_{i=1}^{d'}(\mu(\hat{p}_i)^2+\mu(\hat{q}_i)^2-2\lfloor\mu(\hat{p}_i)\rfloor\lfloor\mu(\hat{q}_i)\rfloor-2\lfloor\mu(\hat{p}_i)\rfloor-2\lfloor\mu(\hat{q}_i)\rfloor-2)\\
&\leqslant \frac{l}{\alpha^2}\sum_{i=1}^{d'}(\mu(\hat{p}_i)^2+\mu(\hat{q}_i)^2-2(\lfloor\mu(\hat{p}_i)\rfloor+1)(\lfloor\mu(\hat{q}_i)\rfloor+1))\\
&\leqslant \frac{l}{\alpha^2}\sum_{i=1}^{d'}(\mu(\hat{p}_i)^2+\mu(\hat{q}_i)^2-2\hat{p}_i\hat{q}_i)\\
&\leqslant \frac{l}{\alpha^2}\sum_{i=1}^{d'}(\mu(\hat{p}_i)-\mu(\hat{q}_i))^2 = LB_{SM}(p,q)\leqslant ED(p,q)
\end{aligned}
$$

□

Theorem 6. *Squared Euclidean distance between two d-dimensional p and q has a lower bound:*

$$
\begin{aligned}
&LB_{PIM\text{–}OST}(p,q)\\
&=\frac{l}{\alpha^2}(\Phi_a(\bar{p})+\Phi_a(\bar{q})-2\lfloor\bar{p}\rfloor\cdot\lfloor\bar{q}\rfloor-2d'-(\Phi_b(\bar{p})-\Phi_b(\bar{q}))^2)
\end{aligned}
\tag{3.12}
$$

where $\Phi_a(\bar{p}) = \sum_{i=1}^{d'}\bar{p_i}^2 - 2\sum_{i=1}^{d'}\lfloor\bar{p_i}\rfloor$ *and* $\Phi_b(\bar{p}) = \sqrt{\sum_{i=d'+1}^{d}\bar{p_i}^2}$ *(resp.*$\Phi_a(\bar{q})$ *and* $\Phi_b(\bar{q})$*).* $\lfloor\bar{p}\rfloor\cdot\lfloor\bar{q}\rfloor = \sum_{i=1}^{d'}\lfloor\bar{p_i}\rfloor\cdot\lfloor\bar{q_i}\rfloor$*. Here, d-dimensional* $\bar{p}$ *is partitioned into two segments of length* d' *and* $d-d'$*.* $\bar{p_i}$ *is the positive floating-point value normalized from* p_i *with* α*, and* $\lfloor\bar{p_i}\rfloor$ *is the integer part.* $\Phi_a(\bar{p})$ *(*$\Phi_a(\bar{q})$*) and* $\Phi_b(\bar{p})$ *(*$\Phi_b(\bar{q})$*) are floating-point values.*

Proof.

$$LB_{PIM-OST}(p,q) = \frac{1}{\alpha}(\sum_{i=1}^{d'}({\bar{p}_i}^2 + {\bar{q}_i}^2 - 2\lfloor\bar{p}_i\rfloor - 2\lfloor\bar{q}_i\rfloor - 2\lfloor\bar{p}_i\rfloor\lfloor\bar{q}_i\rfloor - 2) + (\sqrt{\sum_{i=d'+1}^{d}{\bar{p}_i}^2} - \sqrt{\sum_{i=d'+1}^{d}{\bar{q}_i}^2})^2)$$

$$= \frac{1}{\alpha^2}(\sum_{i=1}^{d'}({\bar{p}_i}^2 + {\bar{q}_i}^2) - 2\sum_{i=1}^{d'}(\lfloor\bar{p}_i\rfloor + 1)(\lfloor\bar{q}_i\rfloor + 1) + (\sqrt{\sum_{i=d'+1}^{d}{\bar{p}_i}^2} - \sqrt{\sum_{i=d'+1}^{d}{\bar{q}_i}^2})^2)$$

$$\leqslant \frac{1}{\alpha^2}(\sum_{i=1}^{d'}{\bar{p}_i}^2 + {\bar{q}_i}^2 - 2\sum_{i=1}^{d'}\bar{p}_i\bar{q}_i + (\sqrt{\sum_{i=d'+1}^{d}{\bar{p}_i}^2} - \sqrt{\sum_{i=d'+1}^{d}{\bar{q}_i}^2})^2)$$

$$\leqslant \frac{1}{\alpha^2}(\sum_{i=1}^{d'}(\bar{p}_i - \bar{q}_i)^2 + (\sqrt{\sum_{i=d'+1}^{d}{\bar{p}_i}^2} - \sqrt{\sum_{i=d'+1}^{d}{\bar{q}_i}^2})^2)$$

$$\leqslant LB_{OST}(p,q) \leqslant ED(p,q)$$

□

Theorem 7. *Cosine similarity between two d-dimensional vectors p, q has an upper bound as following:*

$$UB^{CS}_{PIM-part} = \frac{\lfloor\bar{p}\rfloor\lfloor\bar{q}\rfloor + \Phi_b(\bar{p})\Phi_b(\bar{q}) + \Phi_c(\bar{p}) + \Phi_c(\bar{q}) + d'}{\Phi_a(\bar{p})\Phi_a(\bar{q})} \tag{3.13}$$

where $\Phi_a(\bar{p}) = \sqrt{\sum_{i=1}^{d}{\bar{p}_i}^2}$, $\Phi_b(\bar{p}) = \sqrt{\sum_{i=d'+1}^{d}{\bar{p}_i}^2}$, *and* $\Phi_c(\bar{p}) = \sum_{i=1}^{d'}\lfloor\bar{p}_i\rfloor$ *(resp.* $\Phi_a(\bar{q})$, $\Phi_b(\bar{q})$ *and* $\Phi_c(\bar{q})$*),* $\lfloor\bar{p}\rfloor\lfloor\bar{q}\rfloor = \sum_{i=1}^{d'}\lfloor\bar{p}_i\rfloor\lfloor\bar{q}_i\rfloor$. *Here, d-dimensional* $\bar{p}$ *is partitioned into two segments of length* d' *and* $d-d'$. $\bar{p}_i$ *is the positive floating-point value normalized from* p_i *with* α*, and* $\lfloor\bar{p}_i\rfloor$ *is the integer part.* $\Phi_a(\bar{p})$ *(*$\Phi_a(\bar{q})$*),* $\Phi_b(\bar{p})$ *(*$\Phi_b(\bar{q})$*), and* $\Phi_c(\bar{p})$ *(*$\Phi_c(\bar{q})$*) are floating-point values.*

Proof.

$$UB^{CS}_{PIM-part} = \frac{\sum_{i=1}^{d'} \lfloor \bar{p}_i \rfloor \lfloor \bar{q}_i \rfloor + \sqrt{\sum_{i=d'+1}^{d} \bar{p}_i{}^2} \sqrt{\sum_{i=d'+1}^{d} \bar{q}_i{}^2} + \sum_{i=1}^{d'} (\lfloor \bar{p}_i \rfloor + \lfloor \bar{q}_i \rfloor + 1)}{\sqrt{\sum_{i=1}^{d} \bar{p}_i{}^2} \sqrt{\sum_{i=1}^{d} \bar{q}_i{}^2}}$$

$$= \frac{\sum_{i=1}^{d'} (\lfloor \bar{p}_i \rfloor \lfloor \bar{q}_i \rfloor + \lfloor \bar{q}_i \rfloor + \lfloor \bar{p}_i \rfloor + 1) + \sqrt{\sum_{i=d'+1}^{d} \bar{p}_i{}^2} \sqrt{\sum_{i=d'+1}^{d} \bar{q}_i{}^2}}{\sqrt{\sum_{i=1}^{d} \bar{p}_i{}^2} \sqrt{\sum_{i=1}^{d} \bar{q}_i{}^2}}$$

$$= \frac{\sum_{i=1}^{d'} (\lfloor \bar{p}_i \rfloor + 1)(\lfloor \bar{q}_i \rfloor + 1) + \sqrt{\sum_{i=d'+1}^{d} \bar{p}_i{}^2} \sqrt{\sum_{i=d'+1}^{d} \bar{q}_i{}^2}}{\sqrt{\sum_{i=1}^{d} \bar{p}_i{}^2} \sqrt{\sum_{i=1}^{d} \bar{q}_i{}^2}}$$

$$\geqslant \frac{\sum_{i=1}^{d'} \bar{p}_i \bar{q}_i + \sqrt{\sum_{i=d'+1}^{d} \bar{p}_i{}^2} \sqrt{\sum_{i=d'+1}^{d} \bar{q}_i{}^2}}{\sqrt{\sum_{i=1}^{d} \bar{p}_i{}^2} \sqrt{\sum_{i=1}^{d} \bar{q}_i{}^2}} = CS(p, q)$$

□

Theorem 8. *Pearson correlation coefficient between two d-dimensional vectors p, q has an upper bound as following:*

$$UB^{PCC}_{PIM-part}(p,q) = \frac{d \cdot \lfloor \bar{p} \rfloor \cdot \lfloor \bar{q} \rfloor - \Phi_b(\bar{p})\Phi_b(\bar{q}) + d\Phi_c(\bar{p})\Phi_c(\bar{q}) + d\Phi_d(\bar{p}) + d\Phi_d(\bar{q}) + d \cdot d')}{\Phi_a(\bar{p})\Phi_a(\bar{q})} \quad (3.14)$$

where $\Phi_a(\bar{p}) = \sqrt{d \sum_{i=1}^{d} \bar{p}_i{}^2 - (\sum_{i=1}^{d} \bar{p}_i)^2}$, $\Phi_b(\bar{p}) = \sum_{i=1}^{d} \bar{p}_i$, $\Phi_c(\bar{p}) = \sqrt{\sum_{i=d'+1}^{d} \bar{p}_i{}^2}$, *and* $\Phi_d(\bar{p}) = \sum_{i=1}^{d'} \lfloor \bar{p}_i \rfloor$ *(resp.* $\Phi_a(\bar{q})$, $\Phi_b(\bar{q})$, $\Phi_c(\bar{q})$, *and* $\Phi_d(\bar{q})$*), and* $\lfloor \bar{p} \rfloor \cdot \lfloor \bar{q} \rfloor = \sum_{i=1}^{d'} \lfloor \bar{p}_i \rfloor \lfloor \bar{q}_i \rfloor$. *Here, d-dimensional* $\bar{p}$ *is partitioned into two segments of length* d' *and* $d - d'$. $\bar{p}_i$ *is the positive floating-point value normalized from* p_i *with* α, *and* $\lfloor \bar{p}_i \rfloor$ *is the integer part.* $\Phi_a(\bar{p})$ *(*$\Phi_a(\bar{q})$*),* $\Phi_b(\bar{p})$ *(*$\Phi_b(\bar{q})$*),* $\Phi_c(\bar{p})$ *(*$\Phi_c(\bar{q})$*), and* $\Phi_d(\bar{p})$ *(*$\Phi_d(\bar{q})$*) are floating-point values.*

Proof.

$$
\begin{aligned}
UB^{PCC}_{PIM-part} &= \frac{d\sum_{i=1}^{d'}\lfloor\bar{p}_i\rfloor\lfloor\bar{q}_i\rfloor - \sum_{i=1}^{d}\bar{p}_i\sum_{i=1}^{d}\bar{q}_i + d\sum_{i=1}^{d'}\lfloor\bar{q}_i\rfloor + d\sum_{i=1}^{d'}\lfloor\bar{p}_i\rfloor}{\sqrt{d\sum_{i=1}^{d}\bar{p}_i{}^2 - (\sum_{i=1}^{d}\bar{p}_i)^2}\sqrt{d\sum_{i=1}^{d}\bar{q}_i{}^2 - (\sum_{i=1}^{d}\bar{q}_i)^2}} \\
&\quad \frac{+d\sqrt{\sum_{i=d'+1}^{d}\bar{p}_i{}^2}\sqrt{\sum_{i=d'+1}^{d}\bar{q}_i{}^2} + d\cdot d'}{\sqrt{d\sum_{i=1}^{d}\bar{p}_i{}^2 - (\sum_{i=1}^{d}\bar{p}_i)^2}\sqrt{d\sum_{i=1}^{d}\bar{q}_i{}^2 - (\sum_{i=1}^{d}\bar{q}_i)^2}} \\
&= \frac{d(\sum_{i=1}^{d'}(\lfloor\bar{p}_i\rfloor+1)(\lfloor\bar{q}_i\rfloor+1) + \sqrt{\sum_{i=d'+1}^{d}\bar{p}_i{}^2}\sqrt{\sum_{i=d'+1}^{d}\bar{q}_i{}^2})}{\sqrt{d\sum_{i=1}^{d}\bar{p}_i{}^2 - (\sum_{i=1}^{d}\bar{p}_i)^2}\sqrt{d\sum_{i=1}^{d}\bar{q}_i{}^2 - (\sum_{i=1}^{d}\bar{q}_i)^2}} \\
&\quad \frac{-\sum_{i=1}^{d}\bar{p}_i\sum_{i=1}^{d}\bar{q}_i}{\sqrt{d\sum_{i=1}^{d}\bar{p}_i{}^2 - (\sum_{i=1}^{d}\bar{p}_i)^2}\sqrt{d\sum_{i=1}^{d}\bar{q}_i{}^2 - (\sum_{i=1}^{d}\bar{q}_i)^2}} \\
&\geqslant \frac{d(\sum_{i=1}^{d'}\bar{p}_i\bar{q}_i + \sqrt{\sum_{i=d'+1}^{d}\bar{p}_i{}^2}\sqrt{\sum_{i=d'+1}^{d}\bar{q}_i{}^2}) - \sum_{i=1}^{d}\bar{p}_i\sum_{i=1}^{d}\bar{q}_i}{\sqrt{d\sum_{i=1}^{d}\bar{p}_i{}^2 - (\sum_{i=1}^{d}\bar{p}_i)^2}\sqrt{d\sum_{i=1}^{d}\bar{q}_i{}^2 - (\sum_{i=1}^{d}\bar{q}_i)^2}} \\
&\geqslant \frac{d\sum_{i=1}^{d}\bar{p}_i\bar{q}_i - \sum_{i=1}^{d}\bar{p}_i\sum_{i=1}^{d}\bar{q}_i}{\sqrt{d\sum_{i=1}^{d}\bar{p}_i{}^2 - (\sum_{i=1}^{d}\bar{p}_i)^2}\sqrt{d\sum_{i=1}^{d}\bar{q}_i{}^2 - (\sum_{i=1}^{d}\bar{q}_i)^2}} = PCC(p,q)
\end{aligned}
$$

□

In above PIM-aware bound functions, the term of $\Phi(\bar{p})$ (e.g., $\Phi_a(\bar{p})$, $\Phi_b(\bar{p})$) or $\Phi(\hat{p})$ can be pre-computed. The terms involving dot-product such as $\lfloor\bar{p}\rfloor\cdot\lfloor\bar{q}\rfloor$ and $\lfloor\mu(\hat{p})\rfloor\cdot\lfloor\mu(\hat{q})\rfloor$, are computed by ReRAM PIM. This demands that the data vector $\lfloor\bar{p}\rfloor$ or $\lfloor\mu(\hat{p})\rfloor$, is pre-programmed (written) on crossbars of PIM array, which costs the space of $N\cdot d\cdot b$ or $N\cdot d'\cdot b$ bits, where b denotes the bit size of each operand.

Figure 3.9 shows an example of computing $LB_{PIM\text{-}ED}(p,q)$. At offline stage, the value of $\Phi(\bar{p})$ and vector $\lfloor\bar{p}\rfloor$ are pre-computed, then stored on memory array and programmed on PIM array, respectively. After receiving vector q at online stage, we calculate $\Phi(\bar{q})$ and $\lfloor\bar{q}\rfloor$ once, and reuse them. After PIM generates $\lfloor\bar{p}\rfloor\cdot\lfloor\bar{q}\rfloor$, only the pre-computed value of $\Phi(\bar{p})$ and value of $\lfloor\bar{p}\rfloor\cdot\lfloor\bar{q}\rfloor$ are transferred into CPU. Finally,

the result of $LB_{PIM\text{–}ED}(p, q)$ is cheaply obtained after simple adding and subtracting operations.

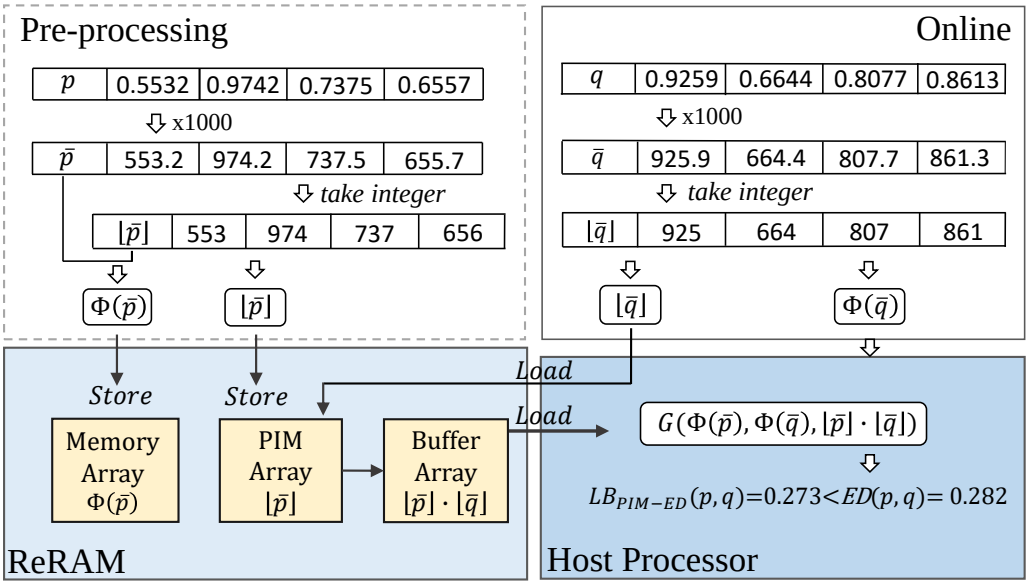


Figure 3.9: Example of computing $LB_{PIM\text{–}ED}(p, q)$.

Recall that α enlarges the normalized values into large values so that we can take the integers. α it impacts the error between $LB_{PIM\text{–}ED}(p, q)$ and $ED(p, q)$, or $LB_{PIM\text{–}FNN}(p, q)$ and $LB_{FNN}(p, q)$. To determine proper α, we have:

Theorem 9. *The error between $LB_{PIM\text{–}ED}(p, q)$ and $ED(p, q)$ has upper bound as:*

$$Error \leqslant \frac{4d}{\alpha} + \frac{2d}{\alpha^2} \tag{3.15}$$

Proof.

$$\begin{aligned} Error &= ED(p, q) - LB_{PIM\text{–}ED}(p, q) \\ &= \sum_{i=1}^{d} ((p_i - q_i)^2 - \frac{1}{\alpha^2}(\bar{p_i}^2 + \bar{q_i}^2 - 2\lfloor\bar{p_i}\rfloor\lfloor\bar{q_i}\rfloor - 2\lfloor\bar{p_i}\rfloor - 2\lfloor\bar{q_i}\rfloor - 2)) \end{aligned}$$

As $0 \leqslant p_i \leqslant 1$, $\lfloor \bar{p}_i \rfloor \leqslant \bar{p}_i = \alpha \cdot p_i$ (resp. q_i), then we have:

$$
\begin{aligned}
Error &\leqslant \sum_{i=1}^{d} ((p_i - q_i)^2 - \frac{1}{\alpha^2}(\alpha^2 p_i^2 + \alpha^2 q_i^2 - 2\alpha^2 p_i \cdot q_i) + \frac{2}{\alpha^2}(\bar{p}_i + \bar{q}_i + 1)) \\
&\leqslant \sum_{i=1}^{d} (\frac{2}{\alpha^2}(\bar{p}_i + \bar{q}_i + 1)) \\
&\leqslant \frac{2}{\alpha} \sum_{i=1}^{d} (p_i + q_i) + \frac{2d}{\alpha^2} \leqslant \frac{4d}{\alpha} + \frac{2d}{\alpha^2} \qquad \square
\end{aligned}
$$

Theorem 9 implies that *Error* is inversely proportional to α. *Error* is inversely proportional to α. Large α makes the bounds tighter. Experimental studies in Section 3.5 illustrate sound tightness of the bounds on real datasets. For example, when using α=10^6, $LB_{PIM-FNN}$ is tight enough to prune 99% of unpromising objects.

3.4.3 PIM Memory Management

PIM-aware bound computation enjoys slight data transfer, but demands to store integer vectors such as $\lfloor \bar{p} \rfloor$ and $\lfloor \mu(\hat{p}) \rfloor$ on crossbars, occupying the space of $N \cdot d \cdot b$ and $N \cdot d' \cdot b$ bits respectively. The typical capacity of contemporary PIM array is only 2GB [60, 81]. PIM array may not have sufficient capacity to accommodate the entire dataset. The simple solution is to divide the dataset into multiple small parts, and each time the crossbars are re-programmed with one part for processing. However, due to the limited write endurance of ReRAM, we should avoid re-programming crossbars. Hence, we propose to compress the dataset based on a given capacity.

Indeed, traditional architecture confronts the same dilemma, and dimensionality reduction techniques employed by the bound functions in Table 3.1 effectively reduce memory usage. The techniques can be adopted for PIM-aware bound functions to decrease the dimensionality from d (d') to s. Then space cost is adjusted to $N \cdot s \cdot b$ bits, where s is the reduced dimensionality. Figure 3.10 depicts an example of computing

$LB_{PIM\text{–}FNN}$ on the compressed s-dimensional vector. Note that compressing the dataset works when the data is integer or floating-point, not binary code.

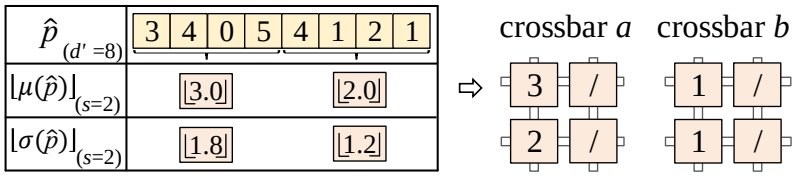


Figure 3.10: Example of reducing the dimension of vector from 8 to 4 (i.e., 2+2) for computing PIM-aware bound.

Theorem 10 establishes the condition of the dimensionality s so that the dataset can fit in PIM array and be processed in the right manner. It suffices to find the maximum value of s such that the approximation of PIM-aware bound obtains the highest possible closeness to the exact value.

Theorem 10. *Given hardware PIM array, and dataset having N d-dimensional vectors, the dimensionality s of compressed vectors is chosen as following conditions:*

$$\textit{Maximize: } s \tag{3.16}$$

$$\textit{subject to: } \begin{cases} n_{data} \leqslant C & s \leqslant m \\ n_{data} + n_{gather} \leqslant C & s > m \end{cases}$$

where $n_{data} = \frac{N \cdot b \cdot s}{m^2 \cdot h}$ is the number of crossbars serving as data crossbar, and $n_{gather} = \frac{N \cdot b}{m \cdot h} \sum_{i=2}^{\sqrt[m]{s}} \frac{s}{m^i}$ is the number of crossbars serving as gather crossbar. C is the number of crossbars in PIM array. Each crossbar contains $m \times m$ cells in h-bit precision. b is the bit size of each operand.

Proof. For easy understanding, we first analyze the crossbar cost for dot-product operation on one pair of vectors. Having two s-dimensional vectors $\lfloor\hat{p}\rfloor$ and $\lfloor\hat{q}\rfloor$, as discussed in Section 3.1, single crossbar can support processing $\lfloor\hat{p}\rfloor \cdot \lfloor\hat{q}\rfloor$ if $s \leqslant m$. The vector $\lfloor\hat{p}\rfloor$ occupies $\frac{s}{m}$ of one crossbar, and all crossbars can serve as *data crossbars*.

Besides, if $s > m$, vector $\lfloor \hat{p} \rceil$ is programmed into multiple data crossbars. In addition to data crossbars, *gather crossbars* are required to aggregate intermediate results. Specifically, dot-product of $\lfloor \hat{p} \rceil \cdot \lfloor \hat{q} \rceil$ costs $\frac{s}{m}$ data crossbars in first cycle, and $\frac{s}{m^2}$ gather crossbars at second cycle to sum up results from first cycle. Hence, $\frac{s}{m^i}$ crossbars are required at i-th cycle, and there are $\sqrt[m]{s}$ depth of cycles in total (the result of division is an integer, otherwise rounded up to integer). Then the number of crossbars that $\lfloor \hat{p} \rceil \cdot \lfloor \hat{q} \rceil$ consumes is as:

$$crossbar(s, m) = \begin{cases} \frac{s}{m} & s \leqslant m \\ \sum_{i=1}^{\sqrt[m]{s}} \frac{s}{m^i} & s > m \end{cases} \tag{3.17}$$

We next present the crossbar cost for the whole dataset having N vectors. Processing one pair of vectors demands $\frac{s}{m}$ data crossbars and $\sum_{i=2}^{\sqrt[m]{s}} \frac{s}{m^i}$ gather crossbars if $s > m$. Moreover, $\frac{s}{m}$ data crossbars actually store vector $\lfloor \hat{p} \rceil$ for $\frac{m \cdot h}{b}$ objects, which can be processed concurrently. Hence, total number of data crossbars and gather crossbars are:

$$\begin{aligned} n_{data} &= \frac{N \cdot b \cdot s}{m^2 \cdot h} \\ n_{gather} &= \frac{N \cdot b}{m \cdot h} \sum_{i=2}^{\sqrt[m]{s}} \frac{s}{m^i} \end{aligned} \tag{3.18}$$

□

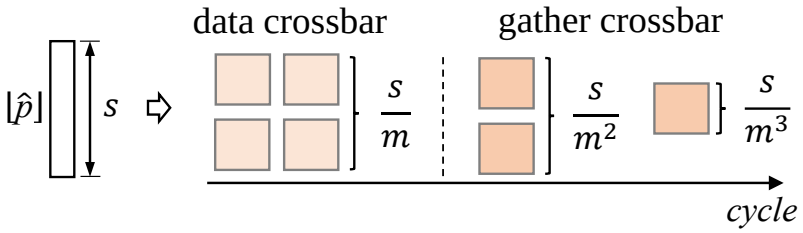


Figure 3.11: Example of crossbar cost for dot-product operation on one pair of vectors (e.g., s = 8, m = 2).

Given PIM hardware and a dataset, s, n_{data} and n_{gather} are confirmed using theorem 10. Then vector such as $\lfloor \hat{p} \rceil$, and all-ones vector $\mathbf{e}$ are programmed to data

crossbars and gather crossbars respectively at offline stage. Besides, recall that we offload the function causing bottleneck to PIM, thus assume that the computation of one PIM-aware bound costs all crossbars of PIM array. Note that it is flexible to separate the crossbars into multiple groups according to practical applications, for parallelly computing multiple functions.

3.4.4 Execution Plan Optimization

Previous subsections present how to process PIM-aware bound function with ReRAM PIM. To exploit ReRAM PIM in an algorithm, we can replace the similarity or bound functions natively running on CPU by their PIM-aware bounds. However, this simple implementation might miss the optimization opportunities to further reduce the unnecessary computation of the algorithm. In this section, we study combining the PIM capability and features of the original algorithm. PIM-aware bound can be inserted at better place of the algorithm, and some existing bounds can be removed.

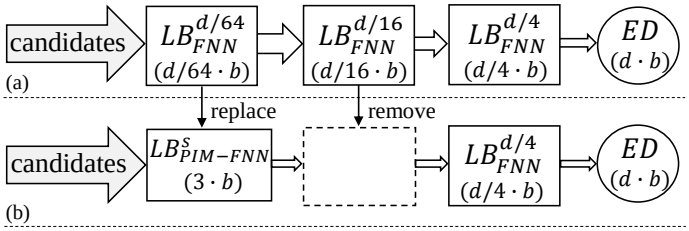


Figure 3.12: Examples of execution plans for FNN algorithm [90]. $(d/64{\cdot}b)$ denotes the data transfer of computing bound for one object is $d/64{\cdot}b$ bits.

The optimization can be achieved by removing some existing bounds. To make it clear, we take kNN algorithm FNN [90] as example. The algorithm FNN applies three bounds based on LB_{FNN} with incremental tightness to progressively prune objects, as depicted in Figure 3.12(a). Here, $LB_{FNN}^{d/64}$ denotes the dimensionality of compressed vector for computing bound is $d/64$. Assuming $LB_{FNN}^{d/64}$ causes the major bottleneck, the computation of which is offloaded to PIM. We thus replace $LB_{FNN}^{d/64}$

by its PIM-aware bound $LB^{s}_{PIM\text{–}FNN}$. Theorem 10 discusses how to choose s with the hardware budget and dataset. $LB^{s}_{PIM\text{–}FNN}$ is used to prune objects, and s is chosen as large as it could be, even larger than $d/64$. Hence, $LB^{s}_{PIM\text{–}FNN}$ might own better pruning power when compared to other original bounds. For example, if $s{>}d/16$, the objects survived from $LB^{s}_{PIM\text{–}FNN}$ are hard to be filtered by $LB^{d/16}_{FNN}$. Removing $LB^{d/16}_{FNN}$ helps to reduce unnecessary computation. If $s < d/4$, $LB^{d/4}_{FNN}$ can remain to filter the objects survived from $LB^{s}_{PIM\text{–}FNN}$.

Assume that the original bounds f_B and PIM-aware bound G compose a *candidate bound set*. There exists a best algorithm that employs a sequence of bound functions (e.g., B_1, ..., B_i, ..., B_g) from the candidate bound set, incurring lowest data transfer cost. Suppose that the candidate bound set has L bounds, then there are 2^L possible algorithm execution plans to enumerate. The dataset has the initial object set D_0 ($|D_0|{=}N$). Objects are gradually filtered by the bounds. After applying bound B_i on D_{i-1}, we obtain a smaller object set D_i. Assume that B_i can prune the objects at percentage of $Pr(B_i)$, we have $|D_i| = N \cdot \prod_{j=1,...,i}(1 - Pr(B_j))$. To identify the best algorithm, we develop a cost equation to estimate the data transfer cost of an algorithm as:

$$Tcost = N \cdot \sum_{i=1,...,g} Tcost(B_i) \prod_{j=1,...,i} (1 - Pr(B_j)), \quad B_i \in (f_B \cap G) \tag{3.19}$$

where $Tcost(B_i)$ denotes the data transfer cost for computing bound B_i. For example, the cost of $LB^{d/64}_{FNN}$ is $d/64 \cdot b$ bits. The least total cost $Tcost$ indicates the best execution plan. As to estimate $Pr(B_j)$, we propose to measure *pruning ratio* of the bound, which has been used in the data search literature [54]. Though PIM-aware bound is indeed processed with PIM, it is practical to run the bound with CPU for purpose of measuring the pruning ratio at offline stage.

3.4.5 Discussion

Our technique has limitation due to the constraint of PIM supported operations. In addition the example functions shown in Table 3.2, as long as one distance function can be transformed as a form that exposes dot-product operation, it is available to adopt PIM-aware decomposition and offload partial computation to PIM. However, when the function does not essentially contain dot-product operation, our technique fails to provide the solution of using PIM. One case is *Jaccard* distance $1 - \frac{|A \cap B|}{|A \cup B|}$, which is defined as the size of intersection divided by the size of union of two sample sets (i.e., set A and B).

We use k-means and kNN as application examples, yet our technique is applicable to data mining applications that involve similarity and dot-product computation as significant component, such as data visualization [158], data cube aggregation [80], data cleaning and reduction [80], frequent pattern mining [79].

3.5 Evaluation

In this section, we present experimental analysis of PIM-optimized algorithms using ReRAM PIM, compared to original algorithms running on conventional architecture.

3.5.1 Experimental Setup

As commercial ReRAM PIM device is still not available, like prior NVM [107, 36, 186] and PIM [105] for database research, we resort to simulation. Specifically, similar to [204, 17], we combine two simulators, a memory simulator NVSim [48] and a system-level tool Quartz [189], to accurately model PIM architecture. NVSim is a prevalent circuit-level simulator based on CACTI [25], which provides a hardware platform to model NVM-based memory such as STT-RAM, ReRAM. We use NVSim to model ReRAM-based memory that enables storage and processing ability, like

Table 3.3: The configuration of hardware platform.

CPU	Broadwell 2.10 GHz Intel Xeon E5-2620; Cache 1/2/3 : 32 KB/256KB/20MB;	
DRAM	16GB DIMM DDR4	
ReRAM-based memory	Memory array	14GB ReRAM
	Buffer array	16MB eDRAM
	PIM array	2GB ReRAM
	Internal bus	50GB/s
ReRAM crossbar	256×256 2-bit precision cells; read/write latency: 29.31/50.88ns; read/write energy cost: 1.08 pJ/3.91nJ	

prior works [36, 169, 81]. Quartz is a software-based NVM performance emulator from Hewlett Packard and has been widely adopted to emulate NVM integrated architectures [222, 149]. It estimates application end-to-end latency by injecting software delays into each epoch. We use Quartz to report system-level performance when assuming main memory is ReRAM.

The configurations of ReRAM PIM and baseline architecture platforms are illustrated in Table 3.3. The only difference between these two architectures is to use ReRAM-based memory or DRAM. The ReRAM-based memory has the same total size as DRAM in baseline platform, i.e., 16GB, in which 2GB is used as PIM array by default. The ReRAM read/write latency and energy cost is 29.31/50.88ns, and 1.08 pJ/3.91nJ respectively, and the parameters are derived from [147, 169]. We follow [60] and configure each crossbar to contain 256*256 cells with 2-bit precision. Then there are default 131072 crossbars to compose PIM array.

We measure the end-to-end execution time of algorithms. The original algorithms are executed on real hardware platform. The execution time of our proposed PIM-optimized algorithms is measured by using NVSim and Quartz. Specifically, NVSim estimates the time of PIM-involved processing executed on ReRAM-based memory,

which includes computing PIM-aware bound on crossbars and buffering PIM results. Quartz is to estimate the time of remaining non-PIM computation in CPU that requires data transfer from memory (including vector data of original dataset stored in memory array, or PIM results in buffer array). Finally, the total execution time of a PIM optimized algorithm is taken as the sum of execution time reported by NVSim and Quartz. We measure the energy consumption for CPU using PAPI [2], and obtain the power usage for PIM from NVSim.

3.5.2 Methodology

Regarding kNN classification, we call the original algorithms as Standard (i.e., linear scan), OST [125], SM [214], and FNN [90]. We name the respective PIM-optimized algorithms as Standard-PIM, OST-PIM, SM-PIM, and FNN-PIM. For PIM-optimized algorithms, we conduct profiling to identify the function causing bottleneck and the ideal performance gain $T_{PIM-oralce}$, as discussed in Section 3.3. Then we follow the techniques in Section 3.4 to offload computation of the function into PIM. Especially, for kNN on *HD*, we only compare Standard and Standard-PIM algorithms.

For k-means, the same initial centers are selected for each experiment. We call the original algorithms as Standard [95], Elkan [56], Drake [49], and Yinyang [46]. We name the respective PIM-optimized algorithms as Standard-PIM, Elkan-PIM, Drake-PIM, and Yinyang-PIM. For each algorithm, we conduct profiling to identify the bottleneck function (i.e., *ED*) and $T_{PIM-oralce}$. Then PIM is used to reduce the distance calculation by providing bound LB_{PIM-ED}.

Table 3.4 lists the real datasets used in our experiments. We normalize the floating-point values into range of [0, 1], and chose α as 10^6 to transform to be non-negative integers. We maintain 32-bit integers for dot-product on crossbars to keep consistent with host processor, and employ the least significant 64-bit of PIM results to avoid overflow. Particularly, the dataset instead is binary vector for kNN on *HD*.

Table 3.4: Statistics of real datasets.

	Dataset	N	d	Size
kNN classification	ImageNet [115]	2340173	150	3.5GB
	MSD [137]	992272	420	2.9GB
	GIST [181]	1000000	960	6.6GB
	Trevi [196]	100000	4096	3.0GB
k-means clustering	Year [184]	515345	90	388MB
	Notre [196]	332668	128	118MB
	NUS-WIDE [37]	269648	500	280MB
	Enron [110]	100000	1369	268MB

We follow [30] to learn 10 million binary codes with length 128-1024 bits from the GIST dataset [181]. We instead take the least significant 32-bit of PIM results on binary vectors.

3.5.3 kNN Classification

We first investigate the performance improvement that ReRAM PIM can contribute to the kNN algorithms. Figure 3.13-3.17 shows the execution time of baseline kNN and PIM-optimized algorithms with the varying datasets, k, and distance functions. We set k=10, distance *ED*, and MSD as default setting and dataset. Standard is the default kNN algorithm. If the PIM array is sufficient to process the dataset, $LB_{PIM–ED}$ is computed with PIM to serve as the bound of exact distance *ED*. When the dataset is too large to maintain in the PIM array, the vector is compressed into the one with dimensionality s as discussed in Section 3.4.3, and we let $LB_{PIM–FNN}$ serve as the bound of exact distance. For example, the dataset MSD is larger than size of PIM array. The data dimensionality is compressed from 420 of the original vector to 105, which is used to compute $LB_{PIM–FNN}$.

Varying datasets: Our PIM framework yields the algorithm acceleration on

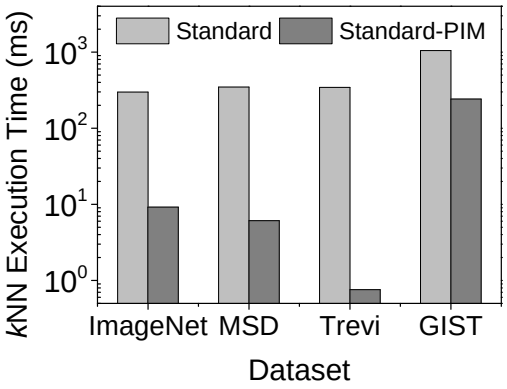


Figure 3.13: kNN classification execution time with varying datasets.

various datasets. Figure 3.13 shows the end-to-end execution time respect to different datasets. Standard-PIM achieves up to 453x speedup compared to Standard. The significant improvement is due to the reduction of data transfer. We observe that the speedup becomes more significant as the increase of dimensionality d of origin vector data. Trevi occurs most significant speedup due to the largest reduction of data transfer: from $4096 \cdot b$ to $3 \cdot b$ bits for each distance computation. Particularly, Standard-PIM shows slight optimization on GIST. This is because the PIM-aware bound $LB_{PIM–FNN}$ is based on LB_{FNN}, and LB_{FNN} natively shows weaker pruning efficiency on GIST than other datasets. For example, using $d'=d/4$ for LB_{FNN} provides average 71.3% approximation of exact distance on GIST, yet 95.4% on MSD.

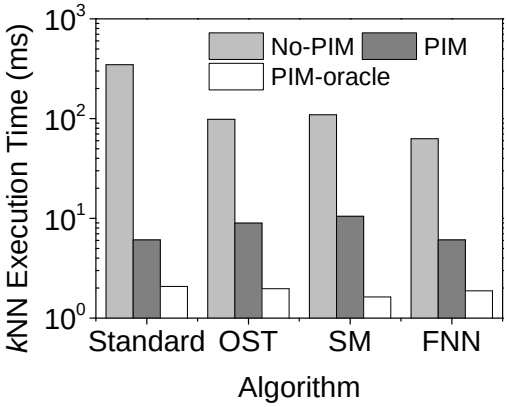


Figure 3.14: kNN classification execution time with varying algorithms.

Varying algorithms: The proposed PIM framework contributes significant improvement on different algorithms. We compare OST, SM, and FNN, to their PIM-optimized algorithms respectively in Figure 3.14. Note that the algorithms are without execution plan optimization proposed in Section 3.4.4. The bottleneck function is replaced by its PIM-aware bound, and other original bounds are still in the algorithms. The state-of-art algorithms are 3.9x faster than Standard on average, and adopting PIM further improves the speedup to 40.8x. Specifically, adopting PIM for algorithm OST, SM and FNN leads to 11.0x, 10.2x, and 10.4x speedup respectively. Moreover, we observe that the performance of PIM-optimized algorithms are close to the optimal gain PIM-oracle. Here, PIM-oracle accounts for the time spent on all other operations except *ED* and bound functions. This implies that our method makes full use of PIM to effectively alleviate the cost of distance computation in the kNN algorithms.

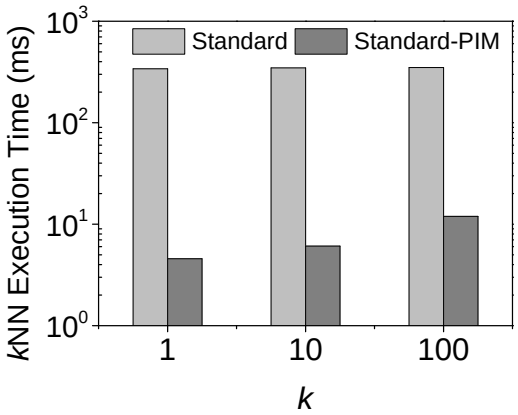


Figure 3.15: kNN classification execution time with varying k.

Varying number of nearest neighbors: The number of nearest neighbors k causes slight impact on the efficiency of our method. Figure 3.15 depicts the execution time respect to different k. Standard-PIM yields 71.5x, 57.1x, and 29.2x speedup compared to Standard respectively. The time ascends slightly with increase of k. This is because the larger k leads to more objects that need exact computation

for candidate refinement.

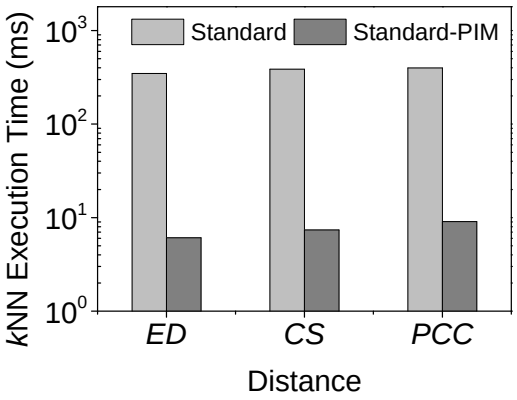


Figure 3.16: kNN classification execution time with varying distance.

Varying distance measures: The proposed PIM framework has stable efficiency on different distance measures. Figure 3.16 shows that the performance gap between Standard-PIM and Standard on three distance measures are close. Especially, we observe that slighter speedup on *PCC* is because that $LB_{PIM-FNN}$ shows weak pruning efficiency on *PCC*, as the computation of both are based on the same statistics - the mean and standard deviation.

Binary vectors: The core contribution of our PIM framework is reducing the computation cost on high-dimensional data, and algorithms operating the data with higher dimensionality can obtain more improvement. Figure 3.17 shows the effect of data dimensionality on PIM-optimized algorithm. We encode the dataset GIST into binary codes of different lengths and choose *HD* as the distance measure. We observe that PIM does not accelerate the algorithm much for 128-bit codes. This is because that computing $HD(p, q)$ with PIM demands loading two dot-product results into host processor, which essentially costs data transfer of 64-bit. Hence, PIM is not a reasonable choice for short binary code. The algorithm benefits from PIM when the dimensionality is high, such as larger than 128-bit. The speedup is significant when the vector is at 1024 dimensions.

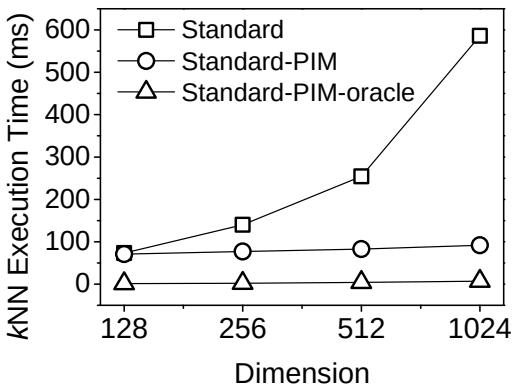


Figure 3.17: kNN classification execution time on binary vector data of varying dimensions.

Energy consumption: Figure 3.18 reports the energy consumption of baselines and Standard-PIM. The power consumption of running algorithms mainly comes from two aspects: processor computation and data transfer. Standard incurs the highest energy cost due to the massive data transfer of entire dataset of all dimensions. OST and SM have lower energy cost because the proposed bounds help alleviate data transfer amount. FNN is the most energy-efficient among baselines, and this is because its bounds have high pruning power. Standard-PIM can save 4.6x energy compared to FNN due to the combined effect of reduced data transfer cost and shortened execution time.

Execution plan optimization: The execution plan optimization (proposed in Section 3.4.4) contributes to the further algorithm acceleration. Figure 3.19 compares the native algorithm FNN and the optimized ones using PIM. FNN-PIM is to accelerate the native FNN by merely using PIM-aware bound. FNN-PIM-optimize denotes the further optimized execution based on FNN-PIM. The result shows that compare to FNN-PIM, FNN-PIM-optimize is closer to FNN-PIM-oracle. This is because FNN-PIM-optimize removes some unnecessary bounds and avoids redundant computation.

We then investigate how the proposed execution plan optimization improves the

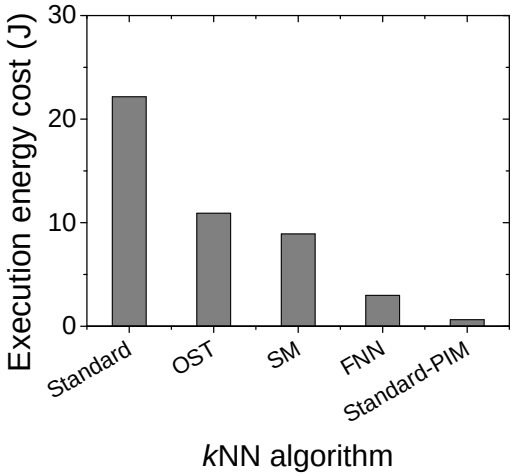


Figure 3.18: Energy consumption of kNN algorithms.

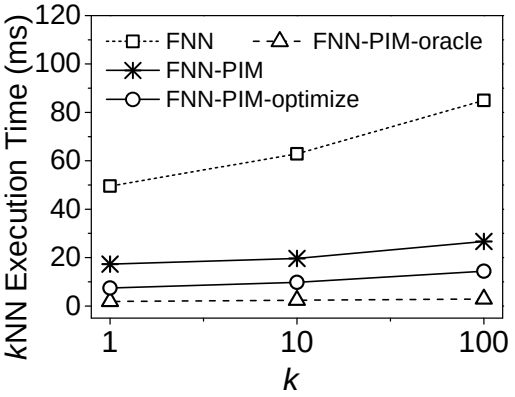


Figure 3.19: kNN classification execution time with execution plan optimization.

algorithm execution. Recall that the native FNN applies three bounds to progressively prune objects (in Figure 3.12). Our profiling results show the bottleneck typically is the first or second bound. FNN-PIM replaces the bottleneck bound by $LB^{s}_{FNN-PIM}$, and keeping other original bounds. Specifically, we found that first bound (LB^{7}_{FNN}) causes the bottleneck. $LB^{s}_{PIM-FNN}$ is thus used to replace LB^{7}_{FNN}, working on filtering objects. Theorem 10 suggests s chosen as 105. At offline stage, we investigated the pruning ratio and data transfer cost of original bounds (e.g., LB^{7}_{FNN}, LB^{28}_{FNN}, LB^{105}_{FNN}) and PIM-aware bound (e.g., $LB^{105}_{PIM-FNN}$), as shown in Figure 3.20. Pruning power of $LB^{105}_{PIM-FNN}$ is stronger than LB^{7}_{FNN} and LB^{28}_{FNN}, and slightly weaker

than LB^{105}_{FNN}. Equation 3.19 suggests that removing all original bounds and only using $LB^{105}_{PIM-FNN}$ leads to least data transfer. The algorithm execution thus avoids the computation cost caused by the three unnecessary bounds.

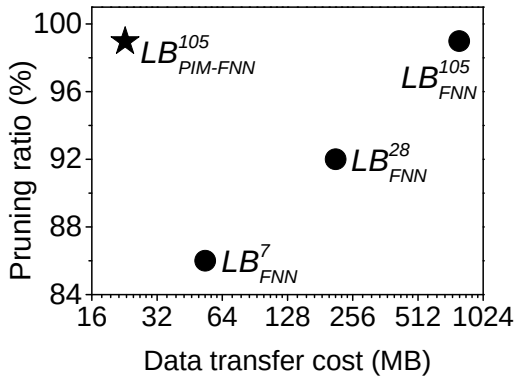
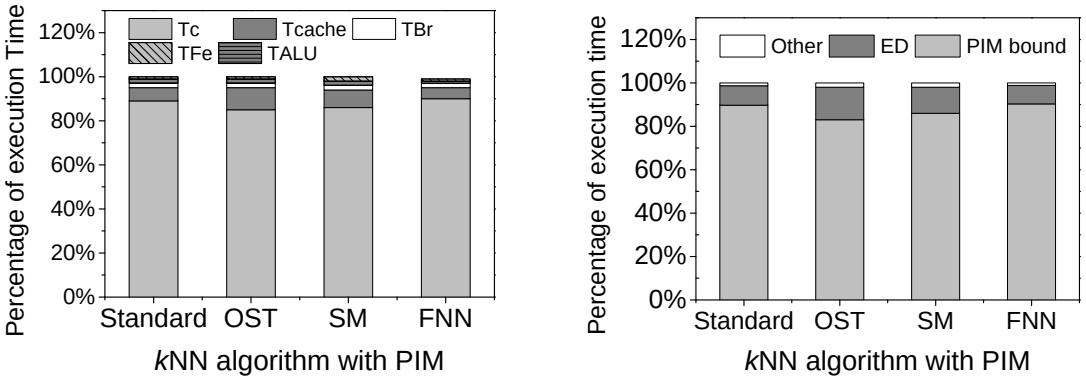


Figure 3.20: Pruning ratio and data transfer cost of computing bound for dataset.

Time breakdown after applying PIM: Recall that kNN algorithm profiling in Section 3.3 by both hardware components and functions indicate that data transfer is the performance bottleneck. We report the time breakdown after applying PIM. T_{cache} is reduced from up 65-83% to 5-9%, and T_c instead dominates the execution time as shown in Figure 3.21(a). Note that T_c includes two parts - the time spending on exact distance by CPU and computing the bounds by PIM. Figure 3.21(b) shows that PIM computation accounts for over 85% of execution time. Specifically, applying PIM to FNN can effectively prune most candidates, and the exact distance computation for survived ones only takes 8.9% of execution time.

Pre-processing cost: we evaluate the pre-processing time cost before the algorithm execution. Pre-processing is an essential procedure for the baselines (except Standard) and PIM-optimized algorithms. The baseline algorithms rely on pre-processing to conduct dimensionality reduction on the data and store it in DRAM for computing bounds. PIM-optimized algorithms pre-compute $\Phi(\bar{p})$ and $\lfloor \bar{p} \rceil$ of PIM-aware bound functions, but instead store in ReRAM-based memory. Figure 3.22 compares the pre-processing time of FNN and FNN-PIM-optimize. The pre-processing



(a) Time breakdown by hardware component (b) Time breakdown by function

Figure 3.21: Performance breakdown of kNN algorithms after applying PIM.

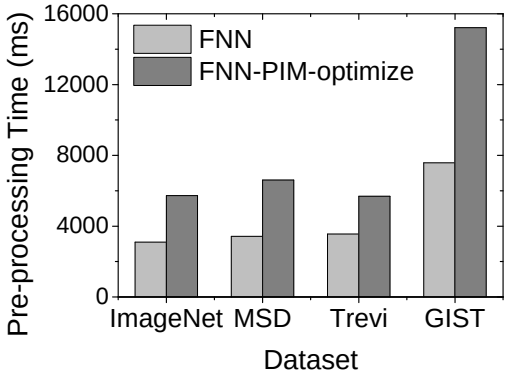


Figure 3.22: Pre-processing time at offline stage for kNN classification.

time of FNN-PIM-optimize is 1.9x slower than FNN on average. The time is impacted by the access speed of memory device and the data amount to write. PIM suffers from longer write latency of ReRAM, but the data amount might be smaller. For instance, FNN needs to prepare three types of vectors for computing LB^{7}_{FNN}, LB^{28}_{FNN}, LB^{105}_{FNN} on MSD, FNN-PIM-optimize prepares only one type for computing $LB^{105}_{PIM-FNN}$, achieving about 33.3% less write access.

3.5.4 k-means Clustering

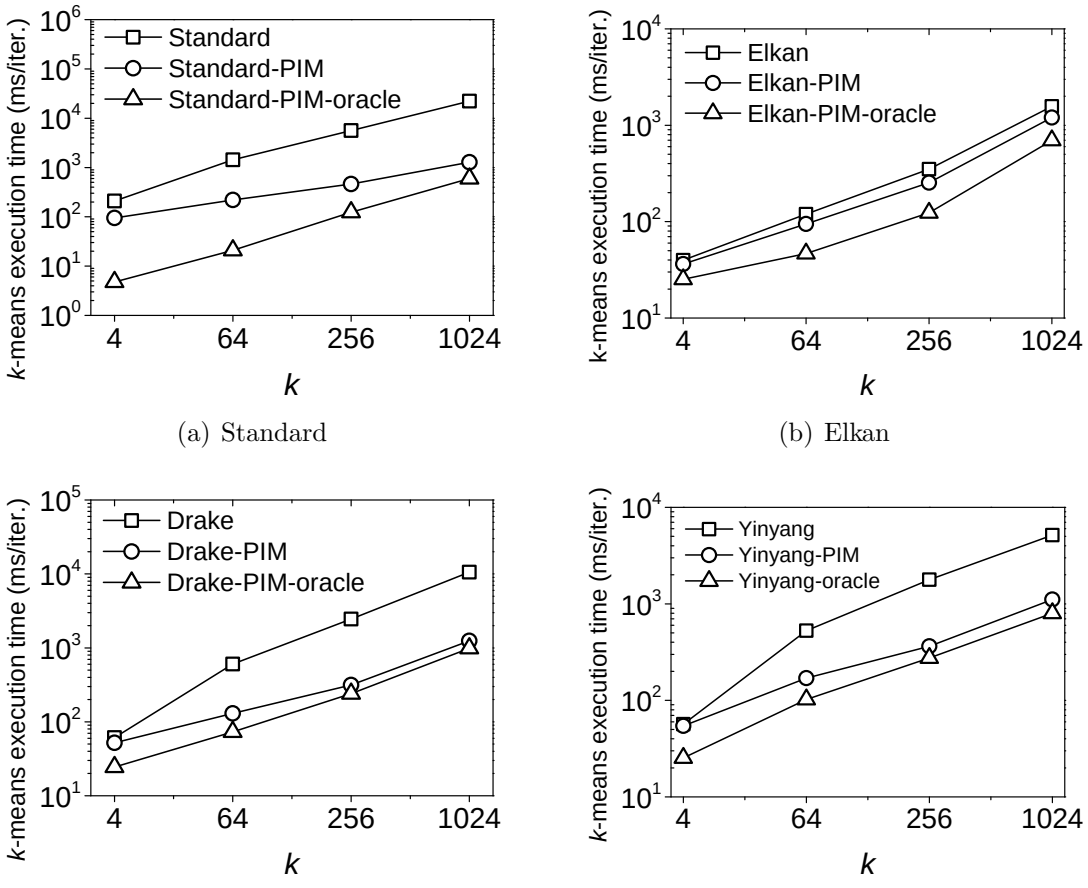
Table 3.5 depicts the performance of k-means algorithms and PIM-optimized ones with varying datasets and the number of centers. *ED* calculation in assign step is

Table 3.5: Execution time on k-means clustering.

Data set	k-means execution time/iteration (ms)								
	k	Stand.	Elkan	Drake	Yinyang	Standard-PIM	Elkan-PIM	Drake-PIM	Yinyang-PIM
Year	4	127.6	46.6	53.2	49.7	36.0	42.4	45.2	**41.6**
	64	478.8	130.3	132.0	119.5	258.2	121.5	74.6	**72.8**
	256	2032.5	398.4	514.8	290.8	684.5	369.5	183.8	**173.3**
	1024	7121.3	1533.5	2151.4	1127.9	1888.7	1364.3	796.8	**539.6**
Notre	4	67.9	34.1	36.6	31.5	36.8	32.3	34.1	**29.7**
	64	549.8	86.6	87.0	110.7	162.2	78.9	**53.8**	70.9
	256	2136.1	258.4	230.0	219.7	425.3	245.5	**117.1**	158.4
	1024	8000.2	1048.1	1453.4	1073.3	1027.2	931.2	**512.4**	574.8
NUS-WIDE	4	153.6	39.8	61.1	56.5	95.0	**37.3**	52.0	54.4
	64	1437.9	119.7	605.8	528.5	218.8	**94.8**	130.4	177.8
	256	5636.9	351.1	2451.2	1778.4	463.9	**253.7**	305.1	364.9
	1024	22273.1	1569.2	10545.3	5957.2	1274.7	**1201.0**	1240.8	1210.5
Enron	4	156.6	20.1	54.8	62.1	49.5	**18.2**	43.6	47.0
	64	1259.6	96.9	447.4	382.5	141.7	**82.8**	85.5	86.9
	256	4764.5	179.6	252.5	759.7	287.7	**162.4**	180.9	187.6
	1024	18984.5	317.2	4102.9	879.6	566.9	**282.6**	486.5	507.3

typically the bottleneck. PIM is used to compute $LB_{PIM–ED}$. The bound contributes to filter far-away centers, and survived ones call exact *ED* calculation. Table 3.5 shows that leveraging PIM yields speedup on all algorithms. This is because expensive *ED* calculation is reduced. The nearest center of each data point is identified faster using PIM-aware bounds. Note that though the overall speedup for k-means clustering is not significant as kNN classification, recall that the goal of our work is to accelerate a given algorithm, rather than a certain application. The experimental results demonstrate the acceleration to each algorithm itself.

PIM gives consistent speedup for Standard, up to 33.4x. Standard severely suffers from heavy data transfer. During assign step, finding the nearest cluster center for data points results in the data transfer of $N{\cdot}k{\cdot}d{\cdot}b$ bits. PIM dramatically decreases it to $N{\cdot}k{\cdot}3{\cdot}b$ bits. With the increase of k or d, the improvement becomes more substantial. Standard-PIM-oracle denotes the theoretical optimal gain with PIM,



(a) Standard

(b) Elkan

(c) Drake

(d) Yinyang

Figure 3.23: Comparison between PIM-optimized and PIM-oracle for k-means clustering algorithms.

i.e., $T_{PIM-oracle}$. $T_{PIM-oracle}$ accounts for the execution time except ED calculation in assign step. Figure 3.23(a) shows the obvious gap between Standard and Standard-PIM, and the narrow gap between Standard-PIM and Standard-PIM-oracle. Higher k makes Standard enjoy greater benefits from PIM.

Elkan-PIM slightly outperforms Elkan, as showed in Figure 3.23(b). This is because ED calculation is not always dominating task for Elkan. Updating original bounds often occupies up to 45% of total time. Elkan maintains k lower bounds for each data point. Though it contributes to prune far-away centers effectively, updating k bounds incurs obvious overhead, leading to high value of $T_{PIM-oracle}$. Elkan-PIM

illustrates an example that PIM might be not considered to be exploited.

Drake-PIM yields the significant speedup when compared to Drake. Not like Elkan, *ED* computation takes the majority of execution time of Drake consistently. As Figure 3.23(c) shows, the gap between Drake and Drake-PIM-oracle is obvious. Drake-PIM achieves up to 8.5x speedup because it bridges the gap effectively, being very close to Drake-PIM-oracle.

Yinyang also enjoys improvement caused by PIM. The significant speedup occurs on high-dimensional datasets, up to 4.9x. To avoid unnecessary *ED* computation, rather than using k lower bounds like Elkan, the global and local filters in Yinyang keep fewer bounds and decrease condition checks. This makes it efficient in clustering low-dimensional data points. However, when the dimensionality is high, the performance degrades fast due to dramatic growth of inevitable *ED* calculation. Figure 3.23(d) shows that Yinyang-PIM mitigates the weakness and eliminates most *ED* calculation, resulting in consistent performance on high-dimensional data.

Figure 3.24 shows that PIM can reduce the energy cost of k-means algorithms in general. The energy saving benefit from PIM follows the trend of execution speedup. Standard-PIM has highest execution speedup, and enjoys the significant energy saving from 2405 to 312J. However, PIM is not effective in reducing the energy cost of Elkan.

Recall that Section 3.3 presents the k-means algorithm profiling by both hardware components and functions. We report the execution time breakdown after applying PIM. Figure 3.25(a) shows that T_{cache} is significantly reduced on the algorithms except Elkan. The reason is that Elkan's bound updating still accounts for the major part of execution time as shown in Figure 3.25(b). T_c dominates the execution time of Standard, Drake and Yinyang after applying PIM. This is because the most computation overhead is producing bounds by PIM. The exact distance computation is effectively alleviated using the PIM bounds. Specifically, Standard-PIM reduces

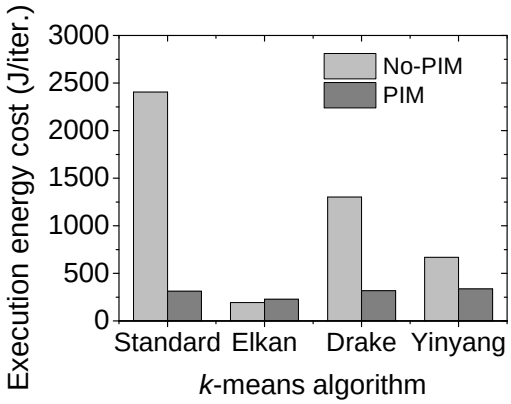
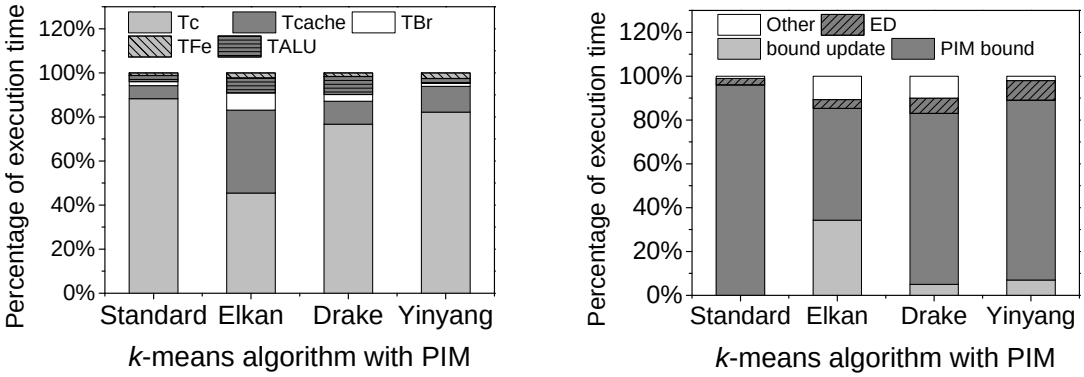


Figure 3.24: Energy consumption of k-means algorithms.



(a) Time breakdown by hardware component (b) Time breakdown by function

Figure 3.25: Performance breakdown of k-means algorithms after applying PIM.

the time on ED computation from over 97% to 6.8% of total execution time.

3.6 Chapter Summary

In this section, we conclude the work presented in this chapter, and discuss the related future research directions.

3.6.1 Conclusion

In this chapter, we propose a novel framework to accelerate similarity-based mining algorithm on high-dimensional data by using ReRAM PIM. PIM is an efficient approach to decrease the substantial amount of data transfer. Previous works have

widely exploited dot-product operation of ReRAM PIM for several applications but not similarity computation.

Our PIM framework solves the several challenges of using ReRAM PIM for similarity-based data mining. First, we conduct the performance profiling to analyze the optimal improvement that the algorithm could expect when adopting ReRAM PIM. Second, we identify PIM-aware function in the algorithm, and then offload most computation of the function into PIM. To combat the limited computational functionality of ReRAM PIM, we propose PIM-aware bound function so that its computation can be accelerated and the accuracy of results won't be compromised. Last, exploiting PIM leads to some computation of the algorithm designed for running on CPU redundant, the further execution plan optimization is proposed to avoid the unnecessary computation and improve the efficiency of PIM. The experiments on kNN classification and k-means clustering indicate that the proposed method is capable of leveraging PIM as modern accelerator to speed up the data mining algorithms.

3.6.2 Research Directions

Emerging NVM PIM has new characteristics different from existing processors and leads to opportunities for further optimization. We consider future research directions as follows.

First, NVM PIM often has a small memory size. When NVM PIM is designed to support processing on high-precision data, the sophisticated circuits such as DAC and S&A, are required to integrate in memory, which makes it suffers from low scalability. To deal with a very large dataset or large intermediate results on NVM PIM of small size, data re-programming on crossbars is inevitable. A space-friendly PIM scheme is thus needed to minimize the impact on re-programming latency and endurance.

Second, it is interesting to examine our techniques beyond similarity-based data

mining tasks. The mining tasks have a wide range of applications. Similarity computation is not involved in many of them, such as decision tree-based classification and grid-based clustering. Exploiting PIM for these tasks introduces new research challenges.

Third, the design of a hierarchical computing system that includes CPU, NVM PIM, GPU and FPGA is an open issue. The modern hardware such as GPU and ASIC can offer the high computing parallelism. The data mining algorithm can offload the partial computation having complex logic to CPU, the part triggering heavy data transfer to NVM PIM, and the part consuming high processing resources to GPU and FPGA.

Forth, our work adopts the common PIM design widely used for many application domains. It is interesting to explore new architectures customized for data mining applications. For example, the organization of crossbars can be redesigned to achieve more effective scatter-gather on high-dimensional vector data. The data mapping onto crossbars might be investigated to support more complex computation such as power mean in Minkowski distance [70] on vector data.

Chapter 4

Accelerating Blockchain Mining by NVM PIM

Blockchain was first proposed in Bitcoin [144] and attracted extensive attentions from both industry and academia. Blockchain could be seen as a public ledger that records all committed transactions with a list of blocks in a decentralized environment. Blockchain has been used in a diverse of fields with various purposes, including Internet of Things (IoT) and social media [38], smart grid networks and edge computing [111]. Blockchain is a decentralized network that avoids single point of failure [43]. As long as the data was added into the chain, the data is always securely stored and can be accessed from any node of the network. Moreover, the data is encrypted with applying time stamping and then added to the blockchain. The source, sequential updates and destination of data are transparent and traceable along the chain [154].

In this work, we focus on the application scenarios that utilize blockchain for *the safe and traceable data storage* [131, 209, 44, 218, 155, 199]. For example, OriginChain [131, 209] is blockchain-based supply chain management system to track product information including originality and components during production, the trajectory during delivery, and the market statistics during sales. [44] designs a

reputation system that records the web browsing trace of users, which helps identify the risky users. [218, 155] use blockchain to store the transactions of remittance and online payment at E-business. Ethereum [199] is a smart contract platform that stores the user transaction records on blockchain.

The applications of using blockchain for data storage often involve a larger number of users in frequent interactions, and thus have to deal with high data write throughput [43]. However, the new data is appended into new blocks, and then the blocks are added into the public chain by performing a *mining* process. This mining process consumes significant energy, time, and computational overhead, which leads to low performance of data storage [199, 16]. For instance, in Ethereum [199], the throughput can only achieve up to 100s of transactions per second [47], and most transactions take over 3 minutes to be included in the chain [170].

Blockchain mining was firstly performed with CPU platforms that suffer from limited repetitive mathematical calculation capability. Then, GPU platforms, benefiting from their highly parallel structure, have replaced CPU to accelerate the mining process. However, GPU platforms experience higher power consumption and higher cost [141]. ASICs, which is customized for specific mining algorithm, have shown their great potential in blockchain mining. Nevertheless, ASIC is application-specific, and cannot be used once the mining algorithm is updated. Thus, ASICs suffer from higher design cost and lower reusability [126]. FPGA, which is programmable hardware that supports reconfiguration after manufacturing, is adopted to assist the mining process [160]. However, it also suffers from unfavored programming difficulty. In this paper, we propose to use ReRAM, which supports in-memory computations, to optimize the blockchain mining.

ReRAM is a kind of emerging non-volatile memory that performs matrix-vector multiplication and sum operation efficiently in a crossbar structure. With ReRAM-based PIM, data movement between memory and CPU is eliminated, thus, releasing

computational resource, saving energy, and reducing latency. In addition, massive crossbars of ReRAM PIM provide high computing parallelism. ReRAM has been widely studied to perform processing-in-memory (PIM) for several kinds of applications. Prime [36] uses ReRAM crossbars to represent synapses and designs a novel PIM architecture to accelerate neural network applications. GraphR [169] and RPBFS [81] are ReRAM-based graph processing accelerators that map graph adjacency matrix to ReRAM matrix-vector crossbar arrays and exploit the massive parallel capability of ReRAM. The integration of applications and ReRAM-based in-memory computing have shown great potential in improving applications' performance.

However, there are several challenges to design a ReRAM-based blockchain mining accelerator since blockchain mining is not a standard matrix-vertex multiplication process. Specifically, 1) $Transformation$: how to transfer blockchain mining to basic ReRAM crossbar multiplication modules. 2) *Data mapping*: how to design efficient mapping schemes to map blockchain transaction data into ReRAM crossbar based modules to perform computations. 3) $Parallelism$: how to efficiently explore the parallelism potentials between blockchain algorithm and ReRAM crossbar design.

To address these challenges, we for the first time propose Re-Mining, a ReRAM-based processing-in-memory architecture for blockchain mining. To transfer blockchain mining process to ReRAM matrix-vector multiplication operations, Re-Mining is designed to consist of a message schedule module (MeS MU) and a SHA computation module (SHA MU). In these modules, we first utilize matrix transformation to transfer rotate right shift (ROR) and right shift (RSF) operations into matrix multiplication operations, then map these matrix operations to ReRAM crossbars and design the ROR and the RSF units. The modules also include an XOR unit, a SUM unit and an AND unit, by extending the corresponding peripheral circuit of ReRAM crossbars. We further propose an intra-transaction parallel framework to accelerate

the SHA computation process for each transaction, and an inter-transaction parallel framework to accelerate the blockchain Merkle tree construction and proof-of-work computation process. The experimental result shows that Re-Mining outperforms 778.5x and 3.8x for blockchain mining process when compared and CPU-based with GPU-based implementation, respectively.

The rest of this chapter is organized as follows. Section 4.1 presents the background and motivation of this chapter. Section 4.2 describes the design and implementation. Experimental results are presented in Section 4.3. Section 4.4 concludes the paper.

4.1 Preliminaries

In this section, we present the background information about blockchain mining process, and discuss the motivation of exploiting ReRAM PIM.

4.1.1 Blockchain Mining

Blockchain works as a decentralized public ledger that stores data, such as records of transactions, with a linked list of blocks. Blocks are generated and shared over the entire blockchain network to prevent system failure, data manipulation and cyberattacks [223]. The foundation of guaranteeing data integrity and validity in blockchain is a computational process – blockchain mining. To generate a new block, a blockchain node, i.e., miner, is required to solve a computing-intensive proof-of-work to obtain a hash value that satisfies a predefined difficulty threshold. After solving the proof-of-work, the result is broadcast to other miners in the networks for validation. The new block is successfully added if the majority of miners agree or reach consensus [206].

Currently, blockchain mining process typically adopts the cryptographic hash algorithm SHA-256 to perform the computation [223]. Our design will also be illus-

trated with the algorithm SHA-256. As shown in Algorithm 4.1, SHA-256 includes a **Message Scheduler Process** that prepares the 512-bit basic computational units and a **SHA Computation Process** that is used to perform the hash operations for these 512-bit computational units.

For the **Message Scheduler Process**, parameters σ_0 and σ_1 are needed. Let $ROR_i(x)$ denote shifting x by i bits rotate right shift, and $RSF_i(x)$ denote shifting x by i bits right shift. $\otimes$ is the XOR operation. We can use the following equations to get these parameters.

$$\sigma_0(x) = ROR_7(x) \otimes ROR_{18}(x) \otimes RSF_3(x) \tag{4.1}$$

$$\sigma_1(x) = ROR_{17}(x) \otimes ROR_{19}(x) \otimes RSF_{10}(x) \tag{4.2}$$

For the **SHA Computation Process**, parameters $\sum_0(x)$, and $\sum_1(x)$, $Ch(x, y, z)$, also $Maj(x, y, z)$ are required. We can use the following equations to get these parameters.

$$\Sigma_0(x) = ROR_2(x) \otimes ROR_{13}(x) \otimes ROR_{22}(x) \tag{4.3}$$

$$\Sigma_1(x) = ROR_6(x) \otimes ROR_{11}(x) \otimes ROR_{25}(x) \tag{4.4}$$

$$Ch(x, y, z) = (x \wedge y) \otimes (\bar{x} \wedge z) \tag{4.5}$$

$$Maj(x, y, z) = (x \wedge y) \otimes (x \wedge z) \otimes (y \wedge z) \tag{4.6}$$

By analyzing these equations, we can conclude that the basic logic computation units for the blockchain mining process consist of ROR, RSF, XOR, SUM, and AND operations. Thus, to accelerate the ReRAM-based blockchain mining, we need to design a architecture that supports the basic logic computation with ReRAM.

Algorithm 4.1 SHA 256

Input: M // a transaction of blockchain

Output: $H_0^N|H_1^N|H_2^N|H_3^N|H_4^N|H_5^N|H_6^N|H_7^N$

1: **Initialize**: Pad and cut M into 32-bits
$(M_1^{(1)}, M_2^{(1)}, ..., M_{16}^{(1)}, M_1^{(2)}, ...M_j^{(i)}, ..., M_{16}^{(N)})$

2: // **Message Schedule Process** for each block

3: **for** j = 0 to 15 **do**

4: $W_j = M_j^{(i)}$

5: **end for**

6: **for** j = 16 to 63 **do**

7: $W_j = \sigma_1(W_{j-2}) + W_{j-7} + \sigma_0(W_{j-15}) + W_{j-16}$

8: **end for**

9: // **SHA Computation Process** for the Message

10: **for** i = 1 to N **do**

11: **for** j = 1 to 64 **do**

12: $T_1 = h_j + \sum_1(e_j) + Ch(e_j, f_j, g_j) + K_j + W_j$
$T_2 = \sum_0(a_j) + Maj(a_j, b_j, c_j)$
$a_{j+1} = T_1 + T_2$, $b_{j+1} = a_j$, $c_{j+1} = b_j$ $d_{j+1} = c_j$
$e_{j+1} = d_j + T_1$, $f_{j+1} = e_j$, $g_{j+1} = f_j$, $h_{g+1} = g_j$

13: **end for**

14: $H_0^i = a^{i-1} + H_0^{i-1}$, $H_1^i = b^{i-1} + H_1^{i-1}$, ...

15: **end for**

4.1.2 Motivation

Figure 4.1 shows examples of computing *ROR*, *XOR*, *SUM*, and *AND* operations with ReRAM. For *ROR* operation, we first transfer a vertex rotate operation to its corresponding matrix-vertex multiplication with Equation 4.7, and then program the modified identity matrix into ReRAM crossbar. Figure 4.1(a) shows that by using the vertex (e.g., 0110) as the wordline input, the output are generated by sensing the current flow (e.g., 1001). Figure 4.1(b) shows an *AND* example with ReRAM.

By modifying the sense amplifier (e.g., SA1), we make it output logical '1' when two cells are both of low resistance status (e.g., 1). Otherwise, SA1 outputs logical '0'. Figure 4.1(c) shows the XOR operation. Similarly, the sense amplifier SA2 is modified to support not only AND but also OR operations, so as to achieve XOR operations [93]. While vertex-matrix multiplication inherently supports SUM operations, to achieve the 32-bits sum operation in blockchain computation, we simply employ the shift and add (S/A) unit to shift and add results of bitlines on ReRAM crossbars [36], as Figure 4.1(d) shown.

$$ROR_2\,(0\ 1\ 1\ 0) = (0\ 1\ 1\ 0)\begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} = (1\ 0\ 0\ 1) \tag{4.7}$$

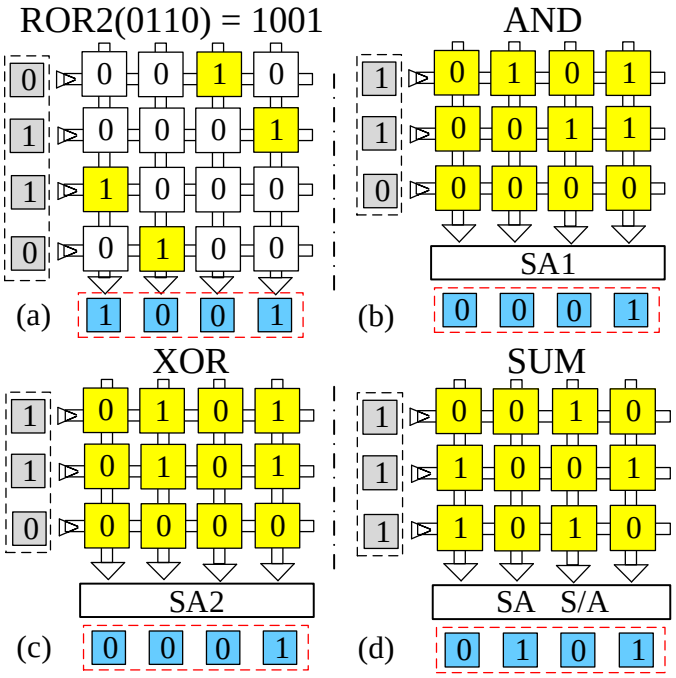


Figure 4.1: Examples of logical operations based on ReRAM.

4.2 Re-Mining: ReRAM Processing-in-memory Architecture for Blockchain

In this section, we first present an overview of the Re-Mining design, and then give the detailed descriptions for each of its function modules.

4.2.1 Design Overview

Figure 4.2 shows the architecture overview. In Re-Mining, each ReRAM bank is partitioned into three regions: memory array, Re-Mining engine, and transaction buffer. The ReRAM memory array serves as conventional memory to store the blockchain transaction data. The Re-Mining engine consists of a Message Schedule Module (MeS MU) and a SHA Computation Module (SHA MU). As shown in algorithm 4.1, both of these two function modules are composed by some basic *ROR*, *RSF*, *XOR*, *SUM*, and *AND* operations. We map these basic logical operations on ReRAM crossbars, and perform the computation with matrix-vertex multiplication. Each Re-Mining engine contains several ReRAM crossbars for the operations during the SHA computation process. Transaction buffer is used to buffers message schedule data (e.g., W_j) and intermediate hash results. Connection bridges data transfer between Re-Mining engine and transaction buffer. The controller is used to decode instructions and generate instructions that coordinate the working process of the Re-Mining engine. The architecture of Re-Mining actually follows the common PIM design that includes three components - storage, computing, and buffer [60, 140].

4.2.2 SHA Computation Module

As shown in algorithm 4.1, when performing the SHA computation process, parameters $\sum_0(x)$, $\sum_1(x)$, $Ch(x, y, z)$, and $Maj(x, y, z)$ are required, and those parameters can be obtained using the equations in Section 4.1.1. We conclude that the SHA computation process is composed by a series of *ROR*, *RSF*, *XOR*, *SUM*, and *AND*

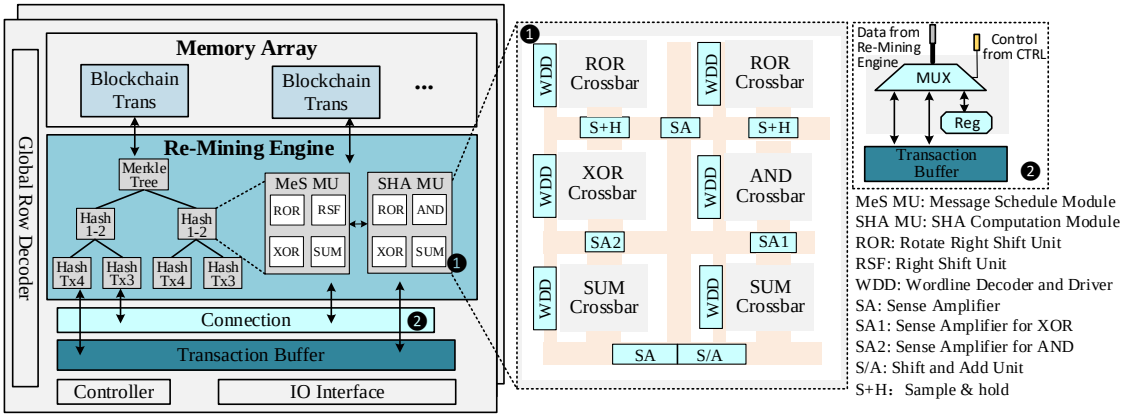


Figure 4.2: Overview of Re-Mining architecture.

operations. In this section, we present the design details of these logical computation units with ReRAM crossbars, and incorporate these basic ReRAM crossbar units to perform the SHA computation.

• **ROR** and **RSF Operation Units**: *ROR* operations occupy the significant computation workload of SHA2. However, *ROR* operations are not inherently matrix-vertex multiplication operations, and cannot be directly mapped to ReRAM crossbars. To design the ReRAM-based *ROR* unit, we first need to transfer *ROR* operations to matrix-vertex multiplication operations. As the example shown in Section 4.1.2, we use Equation 4.7 to transfer the *ROR* operation into a multiplication operation of a vertex and an identity matrix. Then we map the transformed matrix into ReRAM crossbars. The transformation matrix in our *ROR* unit can be abstracted to the following formula:

$$\begin{cases} E_{(i,N+i)} = 0 & (1 \leqslant i \leqslant M-N) \\ E_{(i,i-M+N)} = 1 & (M-N < i \leqslant N) \\ E_{(i,j)} = 0 & (Others) \end{cases} \tag{4.8}$$

Here, $W_{(i,j)}$ is denoted as the cell connected by wordline i and bitline j in a crossbar. M is denoted as the length of the input vector, N represents the number of bits for shift operations. The proposed matrix transformation is based on the property of

identity matrix in linear algebra, $AE = EA = A$, in which E is denoted as an identity matrix, and A is denoted as an arbitrary matrix. We transfer the identity matrix into a new matrix E_{tra} to achieve rotate shift and shift operations: $AE_{tra} = ROR_n(A)$ and $RSF_n(A)$. Our ROR unit can be expended to support right shift, left rotate shift, and left shift operations. For example, for right shift operation, we only need to program cells of $W_{(i,N+I)}$ $(1 \leqslant i \leqslant M - N)$ to be 0.

Figure 4.3 shows an example of the mapping mechanism of our ROR unit. To shift 5 bits for a given 8-bits number '0110 0101', the ROR unit maps a transfered identity matrix E_{tra} into a 8*8 ReRAM crossbar. In our design, E_{tra} is obtained by performing 5-bits rotate right shift of the identity matrix E. So after mapping E_{tra} into the crossbar, the cells $W_{(4,1)}$, $W_{(5,2)}$, $W_{(6,3)}$, $W_{(7,4)}$, $W_{(8,5)}$, $W_{(1,6)}$, $W_{(2,7)}$, $W_{(3,8)}$ are programmed to be logic '1'. '0110 0101' serves as vertex input, and each bit corresponds to one wordline. The worldines that corresponding bit is logical '1' are selected to input discharging voltage, while others are not selected. The result at the output port in the bitlines will be '00101011'.

While above example is discussed by assuming each ReRAM stores one bit data (SLC), our mapping mechanism also apply to ReRAM cells which store multiple bits (MLC). The state-of-the-art technologies used in ReRAM have allowed from 1-bit to 8-bit precision for each cell to support high storage density [220]. Figure 4.3(b) extends the example given in Figure 4.3(a) with ReRAM cells that store 4-bit. The matrix size is minimized to 8*2, so every four 1-bit adjacent cells in ReRAM in each row are integrated into one 4-bit cell.

Benefits of our mapping mechanism for shift operation are three-fold: 1) It supports all kinds of shift operations with arbitrary bits. 2) Once a crossbar for shift operation are programed, it can be utilized repeatedly without consuming more computing resources, which is well-suited for computation that contains tremendous repetitive shift operations, such as proof-of-work in blockchain. In our ROR oper-

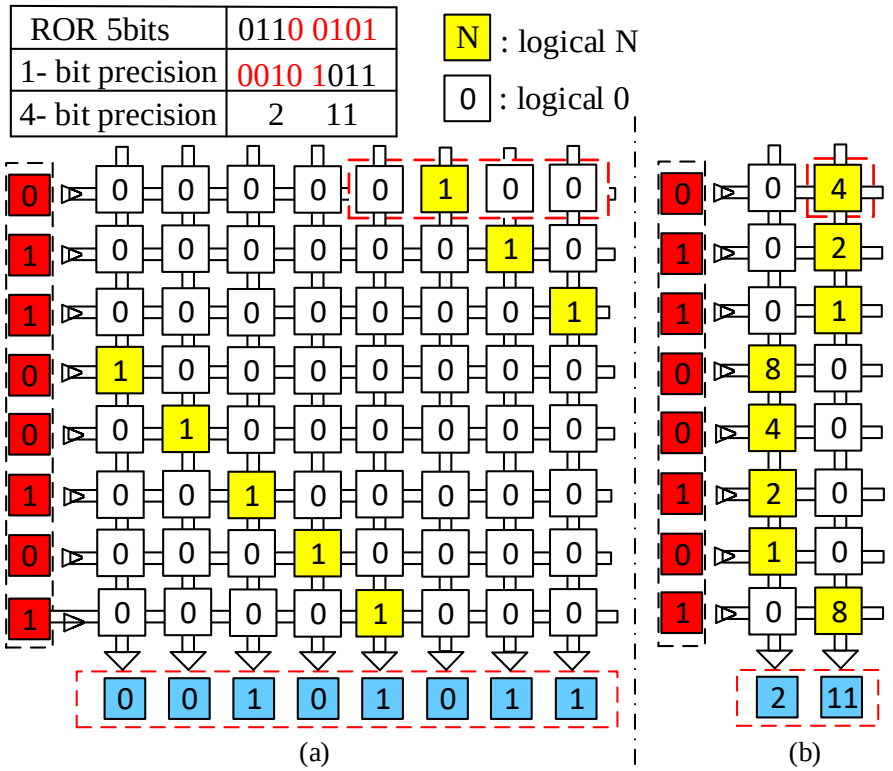


Figure 4.3: An example of ROR operation with ReRAM PIM.

ation unit, all crossbars are merely pre-programmed for one time. 3) Implementing our module in ReRAM only needs to add some simple existing peripheral circuits, such as SA and WDD.

• **AND and XOR operation Unit**: As shown in Section 4.1, *AND* and *XOR* operations occupy the other majority of SHA computation. To support *AND* and *XOR* operations, we modify the sense amplifiers (SAs). *AND* operation in equation Ch and Maj requires two inputs, which results in three situations: if both cells are of high resistance (logical '0'), the bitline current is of low level (I_w); On the contrast, the bitline current is of high level (I_h), when both cells represent logical '1'; If one of two cells is of high resistance (logical '0'), the bitline current is in relatively medium level (I_m). In order to support *AND* operations, we modify the reference value of current detection in SA to output '1' when it detects high current I_h. Similarly, we

modify the reference value of current in SA to output '1' when it detects current lager than I_m to support OR operation. The modifications are achieved by adding two sense resistances R_{AND}, R_{OR} in SAs. XOR operation is implemented based on the results of AND and OR operations. Specifically, the SA outputs '1' when the results of AND and OR operations are '0' and '1', respectively. We adopt the design details of the SA modules from [93]. As Figure 4.2 ① shows, the modified SA1 serves AND operation and the modified SA2 serves XOR operation.

4.2.3 Message Schedule Module

Similarly, when performing the message schedule process, parameters σ_0 and σ_1 are needed, and these parameters can also be obtained using Equation 4.1 and 4.2 in Section 4.1.1. Since there is no AND operation in both equations, the message schedule process are composed by a series of ROR, RSF, XOR, SUM operations, and we also design these logical computation units with ReRAM crossbars. The design details of these units are the same as described in Section 4.2.2, we will omit the design details here.

4.2.4 Intra-transaction Parallelism

The SHA algorithm contains two components: Message Scheduler Process and SHA Computation Process. During these processes, each transaction is divided into 32-bit level computation. In this subsection, we explore the parallelism among different 32-bit computation data sectors to accelerate the transaction SHA process.

First, we note that there is no rigorous data dependency between SHA computation process and message schedule process. SHA computation process requires results from the last cycle of the message schedule process. Thus, these two processes can still work in parallel. Secondly, the 32-bit operands in the equations in Algorithm 4.1 are data-independent with each other. Figure 4.4 shows the paral-

lelism of the message schedule process. To compute W_j, the parameter σ_0 requires W_{j-15} while the parameter σ_1 requires W_{j-2}. Hence, MeS MU can perform ROR and RSF operations in parallel, and then their results can perform XOR operations concurrently. Similarly, during each hash computation loop, equation ch, Maj, $\sum_0$ and $\sum_1$ can be computed in parallel, and their results are used in function T_1 and T_2 to be processed parallelly.

Since intra-transaction parallel is achieved among basic 32-bit operands, the most suitable size of ReRAM crossbars is 32*32 that enables us to reach optimal parallelism. However, ReRAM crossbar size ranges from 4*4 to 1024*1024 [220]. While Re-Mining is also applicable for ReRAM crossbar size which is larger than 32*32, it leads to a trade-off between memory usage and latency when mapping the MeS MU and the SHA MU onto ReRAM crossbars.

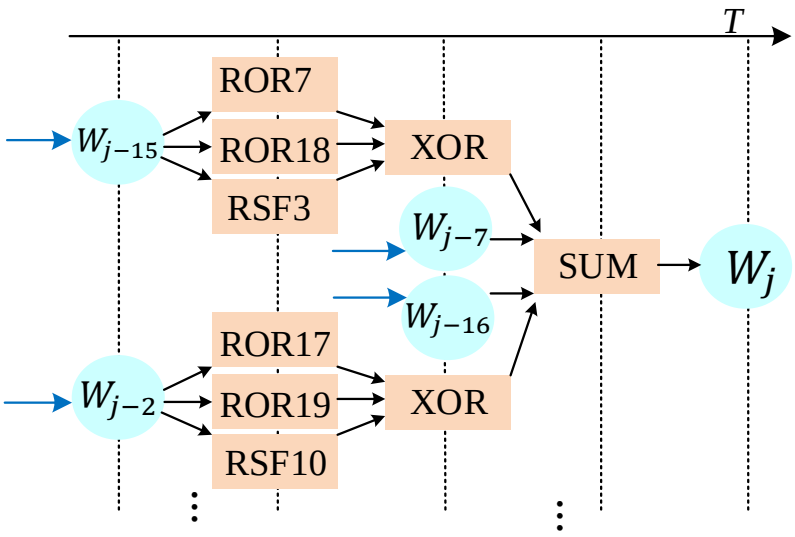


Figure 4.4: Intra-transaction parallel.

Figure 4.5 shows an example of two extremes when mapping the MeS MU module into 64*64 crossbars: 1) the matrices for ROR_7, ROR_{18} operations are mapped into one crossbar in diagonal, while matrices for RSF_3 and ROR_{17}, RSF_{19} and RSF_{10} operations are mapped into other two crossbars, respectively. After W_{j-15} and W_{j-2}

are decoded into wordlines, the function σ_0 and σ_1 can be processed in parallel. 2) mapping all matrices for shift operations in one equation into one matrix needs fewer ReRAM crossbars, whereas it leads to longer latency. Similarly, the trade-off also applies in the SHA MU module.

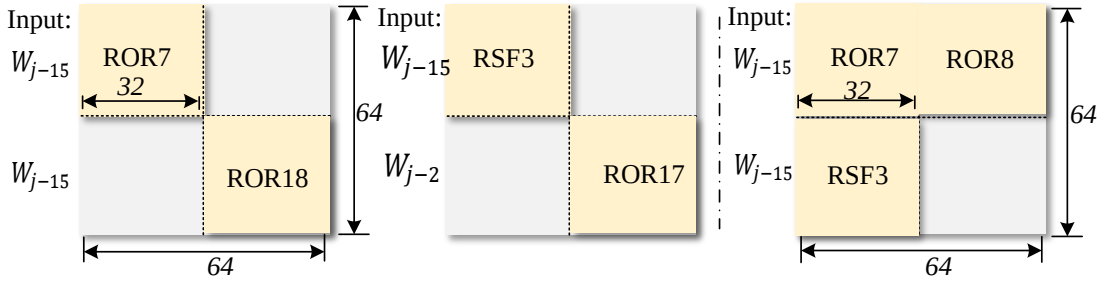


Figure 4.5: Implementation alternatives for 64*64 crossbar.

4.2.5 Inter-transaction Parallelism

As discussed in Section 4.1, the blockchain mining process is separated into two phases: construction of Merkle tree and proof-of-work. In this subsection, we propose the inter-transaction parallelism mechanism to accelerate both of these two phases.

• **Construction of Merkle Tree**: During the construction of Merkle tree, all verified transactions are independent of each other, which can be hashed and decoded in parallel as leaf nodes in the tree with multiple Re-Mining Engines. As shown in Figure 4.2, the leaf nodes in level 1, *HashTx1*, *HashTx2*, *HashTx3* and *HashTx4* are processed in parallel. The results are concatenated and updated as child nodes *Hash1-2*, *Hash3-4* in level 2. Obviously, the children nodes in each level are also independent, and can be processed in parallel with multiple Re-Mining engines. Thus, with more computing engines, Re-Mining achieves a high parallelism level.

• **Proof-of-work**: When we get the root hash value after the construction of the Merkle tree, the proof-of-work phase would start. During the proof-of-work process, the testing process of several nonces could be done simultaneously. Thus, our Re-

Mining architecture will test several nonces with several mining engines in parallel. Having n mining engines means n nonces can be tried concurrently within a single hash latency. The hash value of each testing process is checked then. If it meets the difficulty threshold, a new block is generated and broadcasted to the blockchain network. Otherwise, Re-Mining starts to test another set of nonces with value from $n+1$ to $2n$ until the hash value meets the difficulty threshold.

4.3 Evaluation

4.3.1 Experimental Setup

To evaluate our proposed Re-Mining accelerator, we implement our Re-Mining based on the ReRAM simulation platform NVSim [48]. The ReRAM read and write latency are derived from [169] as $29.31ns$ and $50.88ns$, respectively. The read and write energy cost are 1.08 pJ/3.91nJ respectively. Other related circuit parameters are referenced from [164]. The capacity of ReRAM applied in Re-Mining is of the same size of memory space in GPU (e.g., 11GB). The size of ReRAM crossbars is 64*64. We assume 4-bits MLC for each cell in store array while 1-bit SLC for each cell in Re-Mining array. We utilize 10% of memory capacity for computation. Each computation unit in Re-Mining costs 21 crossbars. For comparison, we also implement the blockchain mining process with CPU and GPU platforms. Table 1 shows hardware specifications of CPU-based and GPU-based platforms. We measure the energy consumption for GPU using NVIDIA system management interface (NVIDIA-SMI) [94], and obtain the power usage for PIM from NVSim. The hardware area usage is also measured by using NVSim.

We extract transaction blocks with varied number of transactions (usually less than 3000) from a most well-known blockchain application - bitcoin [144]. In bitcoin, the number of transactions in each block varies from hundreds to several thousands,

Table 4.1: The configurations of CPU and GPU platforms.

CPU:	Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz
CPU L1 cache	8*32KB
Memory:	16GB
GPU:	NVIDIA GTX 1080i
NVIDIA CUDA Cores	3584
Graphic Memory:	11 GB GDDR5

but generally less than 3000. The computation in blockchain mining process mainly includes two parts: hash of Merkle tree and proof-of-work. We firstly show the performance of Re-Mining when computing the root hash value of Merkle tree. Then, we compare Re-Mining with CPU and GPU to show the performance improvement for proof-of-work and throughput under different difficulties.

4.3.2 Micro Performance

We randomly select 5 blocks with increasing sizes, which contain around 1000, 1500, 2000, 2500, 3000 transactions, respectively, from bitcoin as the workloads. Figure 4.6 shows the performance comparison of Re-Mining, CPU, and GPU in terms of latency (ms). Generally, Re-Mining has the lowest latency across the board. With the number of transactions in the blocks increasing, the latency also increase, but Re-Mining suffers the lowest incremental rate. When the number of transactions in blockchain blocks is increased to 3000, Re-Mining outperforms 80.1x than CPU-based implementation, and 2.6x than GPU-based implementation.

4.3.3 Macro Performance

The performance of Re-Mining for proof-of-work computation in blockchain is depicted in Figure 4.7. We vary the difficulties required by blockchain algorithm from

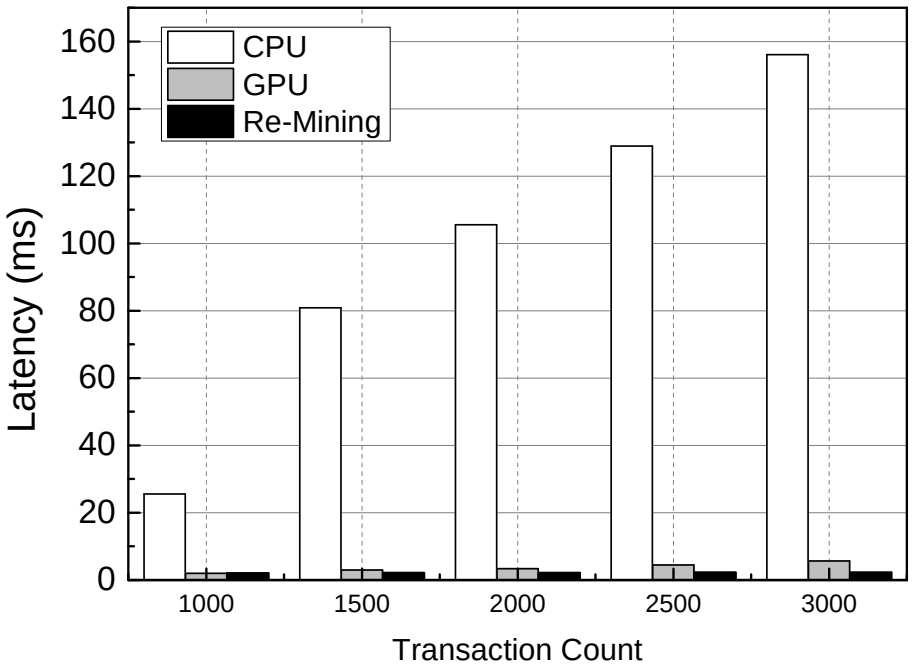


Figure 4.6: Running time for Merkle tree computation.

5 to 9 zero bits. Our proposed Re-Mining accelerator outperforms up to 778.5x than CPU-based implementation, and up to 3.8x than GPU-based implementation. The reason is that our Re-Mining architecture reduces the data movement overhead between processing unit and memory, and achieves high parallelism by testing multiple nonces in parallel during poof-of-work process.

Compared with improvement of micro workloads, the performance improvement of macro workloads is much higher. This is because the volume of data in each block is limited, the computation of which is far away to reach the full potential parallelism of GPU and Re-Mining. On the other hand, the proof-of-work process in blockchain requires a huge amount computation, which could better utilize the parallel processing capability of our Re-Mining architecture.

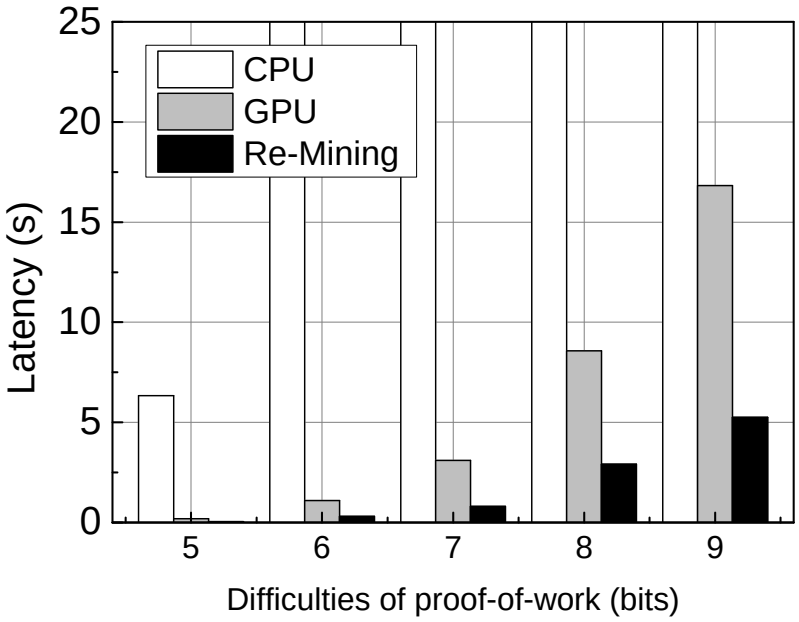


Figure 4.7: Running time for proof-of-work computation.

4.3.4 Throughput Performance

Figure 4.8 shows the throughput performance (the number of transactions per second) of Re-Mining on varying difficulties. CPU has the lowest throughput, and yields below 1 TPS when the difficulty has 9 zero bits. GPU shows higher throughput than CPU, reaching over 15K when the difficulty has 5 zero bits, but significantly dropping at around 110 when the difficulty has 9 zero bits. Re-Mining achieves the optimal performance, showing up to 3.6x higher throughput than GPU. The throughput performance follows the observation of macro performance, since the mining process is dominated by proof-of-work computation. The throughput performance of Re-Mining demonstrates its efficiency to improve data storage in blockchain applications.

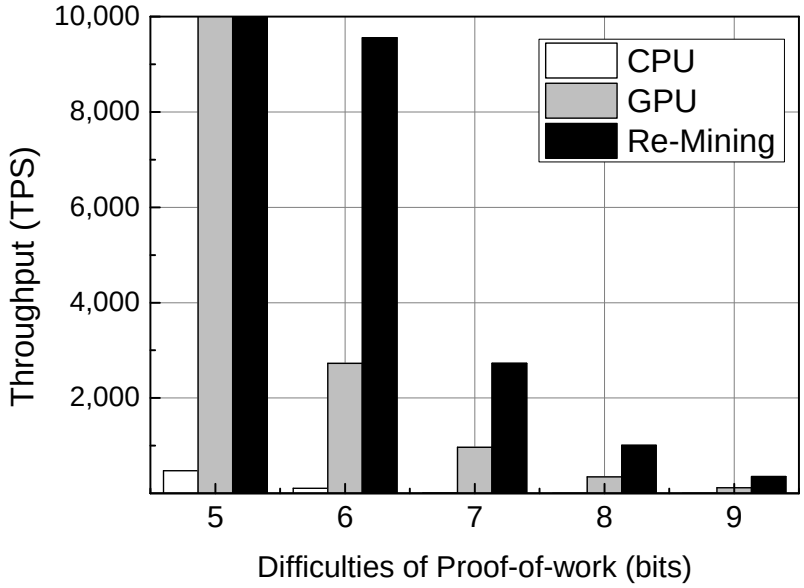


Figure 4.8: Performance for transaction throughput.

4.3.5 Energy Consumption

Figure 4.9 reports the energy consumption of Re-Mining on varying difficulties. CPU has lowest average power (185 watts) on mining computation, while the longer execution time leads to the highest energy cost. Re-Mining has higher average power, but still achieves energy reduction due to the reduced execution time. Specifically, Re-Mining achieves up 2.6x energy saving compared to GPU when the difficulty has 9 zero bits. The energy consumption of CPU-based and GPU-based implementations mainly consist of two aspects: processor computation power and memory access power. Re-Mining eliminates the data access and shortens execution time, and thus consumes less energy compared to CPU and GPU.

4.3.6 Hardware Area

We compare Re-Mining to CPU and GPU on hardware area. The CPU's chip area is 912 mm^2 [185, 60] and GPU's chip area is 471 mm^2 [159, 171]. Re-Mining has three components - memory array, Re-Mining engine and transaction buffer, of which

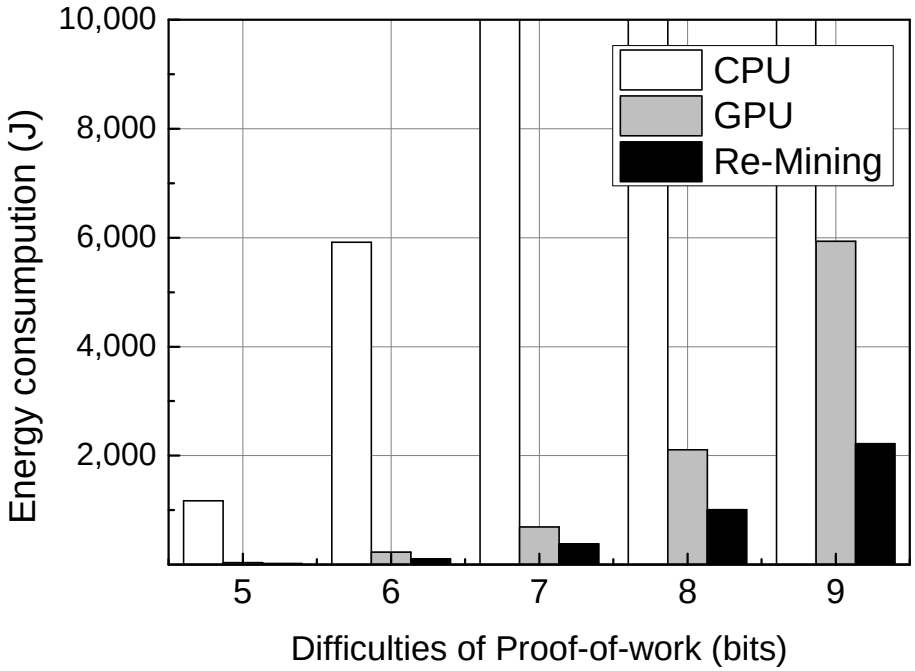


Figure 4.9: Energy consumption of proof-of-work computation.

Re-Mining engine is the part that works like CPU and GPU to process data. The simulation results show that Re-Mining engine is more area-efficient than GPU and CPU, and only takes around 452 mm^2 area. The circuit that consumes the largest area is ADC to support dot-product computation. If we include the area usage of memory array and transaction buffer, the area of Re-Mining is 1.3x larger than CPU and 2.5x than GPU. The area is expected to decrease effectively with advance 3D NVM stacking [156] and multi-level storage [217] techniques.

4.4 Chapter Summary

In this section, we conclude the work presented in this chapter, and discuss the related future research directions.

4.4.1 Conclusion

In this chapter, we propose Re-Mining, a novel ReRAM PIM accelerator for blockchain mining process. The blockchain mining algorithm is mapped on ReRAM crossbars with efficient parallelism. We transform the logical operations such *ROR*, to matrix-vertex multiplication supported by ReRAM PIM. We propose the parallelism optimization to make full use of ReRAM crossbar resources. Experimental results show that Re-Mining outperforms GPU-based and CPU-based implementations significantly with both micro and macro workloads. The accelerated mining can contribute to fast data storage in blockchain systems. Re-Mining includes processing cryptographic hash (i.e., SHA-256) as main component. Beyond the Blockchain, cryptographic hash function is also widely used in many application domains such as image encryption [59, 72], web server [136] and RFID [64], which can adopt our design to accelerate the hash computation.

4.4.2 Research Directions

To exploit NVM PIM for blockchain applications, we consider future research directions as follows.

First, high energy consumption of blockchain mining is a typical issue for industry applications. One attractive characteristic of NVM PIM is lower energy cost than the processors including CPU and GPU. The straightforward way to decrease energy cost is adopting fewer memory crossbars for processing, which yet leads to lower computing parallelism. How to balance the trade-off between computing parallelism and the energy consumption is significant to further improve practicality of NVM PIM. Recent NVM simulators [48, 203, 32] provide the functionality to model energy consumption of PIM designs.

Second, the blockchain is trusted only if over 51% computing resources is con-

trolled by trusted parties. When NVM PIM is ready to be involved in a blockchain network, NVM PIM as new processor participates mining new blocks, cooperating with other processors such as GPU and ASIC in manner of heterogeneous computing. Various processors have different properties and computing power. One issue arises that how to ensure over 51% computing power is always occupied by NVM PIM, and at same time, to achieve high performance of mining by making full use of all processors.

Chapter 5

Speeding Up End-to-end Query Execution Via Learning-based Progressive Cardinality Estimation

Query optimizers of database systems rely on the estimated cardinalities to evaluate the costs of the execution plans, and can find the optimal execution plan if the estimations are error-free [122]. Traditional approaches exploit statistics of the tables (e.g., via histogram or sampling) for cardinality estimation [201, 130, 45, 5]. Their errors are large (e.g., many orders of magnitude) as they fail to account for the correlations among the tables [130, 122]. Recently, many learning-based cardinality estimation methods have been proposed [109, 175, 195, 213], which significantly improve the accuracy of traditional methods. These learning-based estimators often involve complex model designs that trigger higher computation overhead than traditional approaches. Modern GPUs provide high computing parallelism, thus they are widely used in the learning-based estimators [109, 212, 195, 84, 127, 200, 175].

Existing learning-based cardinality estimators can be classified into three categories [200, 195]: *data-driven*, *query-driven* and *hybrid*. Data-driven estimators [85, 82, 212] regard cardinality estimation as an unsupervised learning problem and model the joint distribution of the relation tables. Query-driven estimators [109, 175, 53]

Table 5.1: A comparison of some learning-based cardinality estimators.

	Estimator	Data access	q-error for estimation	Inference time (ms)
Hybrid	UAE [200]	Yes	5.05	20.6
Data-driven	NeuroCard [212]	Yes	6.39	20.2
	DeepDB [85]	Yes	8.62	29.5
Query-driven	MSCN [109]	No	54.1	0.01
	TLSTM [175]	No	39.8	0.52
	LPCE (our work)	Yes	15.7	0.27

treat cardinality estimation as a regression problem that maps feature vectors extracted from the query content to cardinality values. Hybrid estimators [200] learn from both data and query for better accuracy.

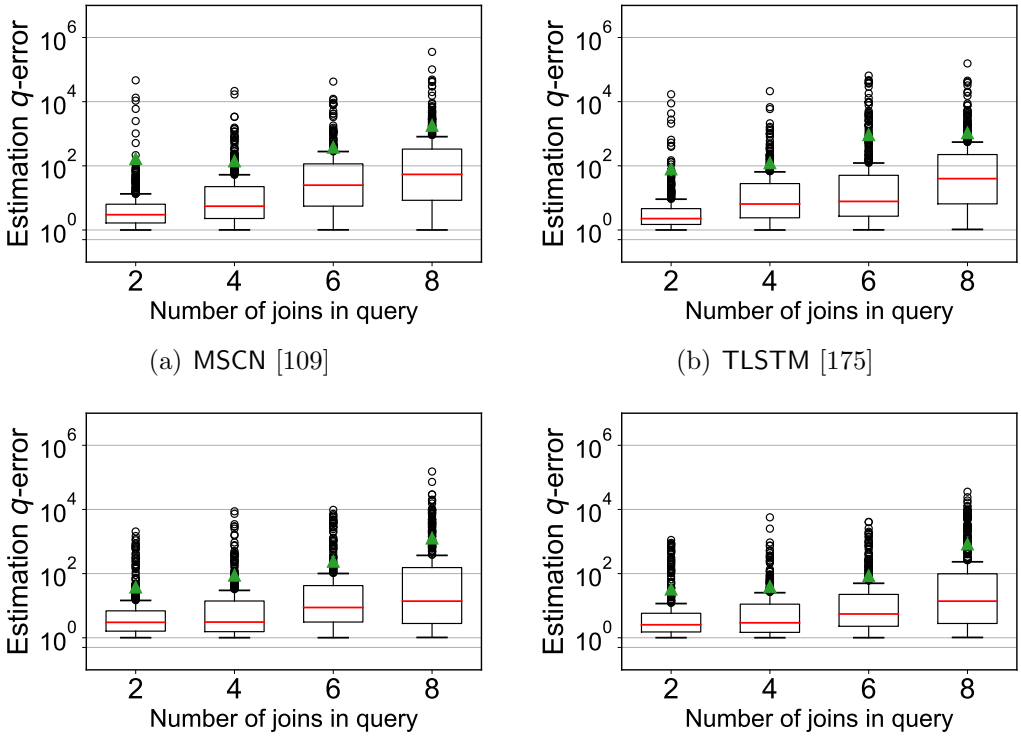
In this work, we focus on query-driven estimators for three reasons. First, they can be easily integrated into existing database systems in a fashion we call *model-as-a-service*. They require only query samples and result cardinalities, and thus model training and inference can be provided as a service (e.g., on the cloud). The database system and query optimizer are regarded as black boxes, and thus query-driven estimators can be upgraded transparently. This is important given the rapid development of learning-based estimators (e.g., from MSCN [109] in 2019 with about 600 LoC to NeuroCard [212] in 2021 with 8000+ LoC). In addition, their training and inference costs do not have to scale with data size, and hence are more friendly to large-scale data. Second, query-driven estimators do not access the relational tables, and thus are free from data security problems, which are important for areas such as finance and medicine [12]. Third, query-driven estimators typically have shorter inference time than data-driven and hybrid estimators, as we will show shortly in Table 5.1, and thus add less overhead to end-to-end query execution time.

To understand the trade-offs of existing learning-based estimators, we conduct a preliminary test on the IMDB dataset [122]. We generate a set of queries with 8

joins by following [109], and report the estimation error and average inference time of existing learning-based estimators in Table 5.1. The results show that query-driven estimators have larger errors but shorter inference time than data-driven and hybrid estimators. This can be partially explained by the fact that query-driven estimators use less information than data-driven and hybrid estimators. To investigate the effect of query complexity (e.g., the number of joins) on existing learning-based estimators, we generate a set of queries whose number of joins ranges from 2 to 8. We plot the estimation error of different estimators in Figure 5.1. Figures 5.1(a) and (b) show that the errors of query-driven estimators are small for simple queries (e.g., with 2 or 4 joins) but become large for complex queries with more joins (e.g., >100x for 8 joins). One reason is that estimation errors propagate and amplify when more operators are involved. Guided by inaccurate estimations, the initial execution plan found by the query optimizer via query-driven estimators can be far from optimal, especially for complex queries.

Inspired by the observations above, we propose a novel query-driven estimator, i.e., *progressive cardinality estimator*, which dynamically refines *cardinality estimations for the remaining operators* in the execution process using *the exact cardinality of the executed operators*, such that the query optimizer can adjust the execution plan to reduce end-to-end execution time. Thus, cardinality estimation enjoys the short inference time of query-driven estimators and high accuracy at the same time. Interestingly, we found that data-driven and hybrid estimators also have larger errors for complex queries, as shown in Figures 5.1(c) and (d). We plan to extend our progressive cardinality estimation framework to them in the future.

We provide one schematic illustration of progressive cardinality estimation in Figure 5.2, where the left panel shows the initial execution plan. After executing the operator at bottom of the join tree (i.e., $\mathtt{R}_\sigma \bowtie \mathtt{T}_\sigma$), we obtain its exact cardinality (i.e., 1,128), which enables more accurate cardinality estimations for sub-plans



(a) MSCN [109] (b) TLSTM [175]

(c) NeuroCard [212] (d) UAE [200]

Figure 5.1: Cardinality estimation accuracy for the queries with different number of joins.

that involve the result of $\mathtt{R}_\sigma \bowtie \mathtt{T}_\sigma$ (e.g., $(\mathtt{R}_\sigma \bowtie \mathtt{T}_\sigma) \bowtie \mathtt{U}_\sigma$, $(\mathtt{R}_\sigma \bowtie \mathtt{T}_\sigma) \bowtie \mathtt{S}_\sigma$, and $(\mathtt{R}_\sigma \bowtie \mathtt{T}_\sigma) \bowtie \mathtt{S}_\sigma \bowtie \mathtt{U}_\sigma$). With the refined estimations, the optimizer may find that the plan in Figure 5.2(b) is better, whose *join order* (for table $\mathtt{U}_\sigma$ and $\mathtt{S}_\sigma$) and *operator execution method* (from hash join to nested loop join) are different from Figure 5.2(a). Progressive cardinality estimation poses challenges on both the effectiveness and efficiency of the estimator. Specifically, effectiveness means that the estimator should fully utilize the information of the executed operators to provide more accurate cardinality estimations for the remaining operators. Efficiency means that the estimation model should have a short inference time as the overhead of progressive estimation adds to end-to-end query execution time.

To address the challenges above, we propose a framework named Learning-based

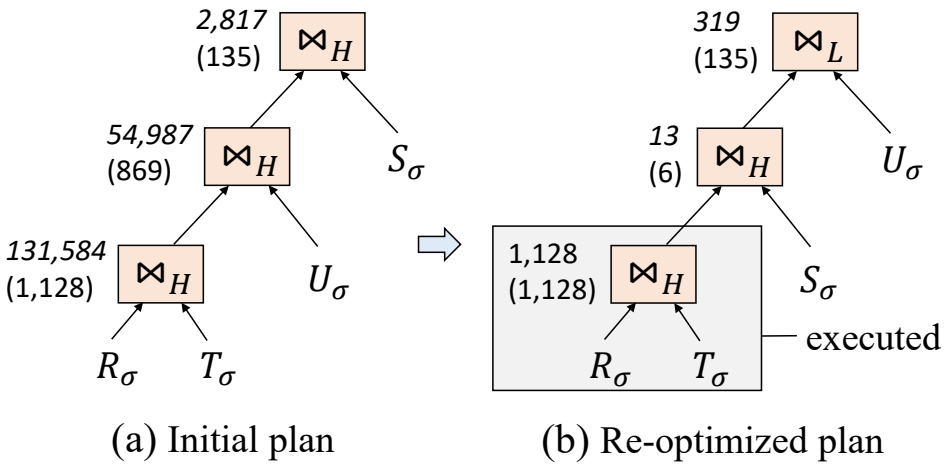


Figure 5.2: Query re-optimization example, upright number for real cardinality and italic number for estimated value. $\bowtie_L$ indicates nested loop join and $\bowtie_H$ is hash join.

Progressive Cardinality Estimation (LPCE). It consists of an initial cardinality estimation model (LPCE-I) and a progressive cardinality refinement model (LPCE-R).

LPCE-I is designed to provide accurate initial estimation with low cost. In particular, LPCE-I utilizes a *node-wise loss function* to learn from the internal operators in an execution plan (instead of only the final query result as in existing estimators), which leads to high estimation accuracy. In addition, LPCE-I employs *a SRU-based model backbone*, and thus enables fast inference as SRU has fewer parameters than LSTM used in existing query-driven estimators and enjoys parallel matrix computation. Last but not least, we use a *knowledge distillation-based model compression* technique to further improve the inference efficiency of LPCE-I. The SRU-based design and model compression lead to short inference time on GPU.

LPCE-R is designed to progressively refine cardinality estimations for the remaining operators in the execution process. LPCE-R consists of three modules: one module is trained to extract the query contents of the executed operators, the second module focuses on the exact cardinalities of the executed operators, while the third module fuses the information provided by the former two modules to conduct estimation for the remaining operators. All three modules adopt the lightweight structure of

LPCE-I, and thus LPCE-R also enjoys efficient inference.

The reasons that we choose GPU instead of NVM PIM for cardinality estimation are two-fold: 1) Advances database engines such as HeavyDB [83], Kinetica [108] and BlazingSQL [18], have already adopted GPU as accelerator to speedup query execution. Instead of introducing new hardware, it is cost-effective to further explore utilizing GPU for cardinality estimation. 2) Existing learning-based estimators (e.g., DeepDB, NeuroCard) are widely used with GPU, though they cannot achieve accuracy and efficiency simultaneously. LPCE on GPU can meet the urgent needs of user to shorten query end-to-end execution time with both accurate and effective cardinality estimation, by easily replacing the estimator without waiting for a ready commercial NVM PIM device. We leave the implementation issues on NVM PIM as future work.

To summarize, we made the following contributions in this work:

- We observe that the estimation errors of query-driven cardinality estimators increase with query complexity and propose progressive cardinality estimation to combat estimation errors and reduce end-to-end query execution time.
- We design the LPCE framework for progressive cardinality estimation, which consists of an initial estimation model (LPCE-I) and a progressive refinement model (LPCE-R). LPCE-I adopts node-wise loss function, SRU-based backbone and knowledge distillation to achieve accuracy and efficiency (Section 5.2). LPCE-R uses a structure with three modules to effectively extracts information from the executed sub-plans and accurately refines the cardinality estimations for the remaining operators (Section 5.3).
- We integrate LPCE into PostgreSQL, a well-known SQL engine (Section 5.4) and conduct extensive experiments on the widely used IMDB dataset (Section 5.5). The results show that LPCE outperforms recent query-driven, data-driven and hybrid estimators in end-to-end query execution time because of its high estimation

accuracy, fast inference and progressive estimation refinement.

The remainder of the paper is organized as follows. Section 5.1 introduces the overall framework and design goals of LPCE. Section 5.2 presents our initial cardinality estimation model LPCE-I. Section 5.3 discusses the design of our progressive cardinality refinement model LPCE-R. Section 5.4 describes how we integrate LPCE into PostgreSQL. Section 5.5 reports our extensive experimental evaluation. Section 5.6 reviews the most relevant works while Section 5.7 draws the concluding remarks.

5.1 Overview of the LPCE Framework

We consider *select-project-equijoin-aggregate* [134, 109] queries in the following form:

$$\begin{aligned}&\mathsf{SELECT}\quad \mathsf{COUNT}(*)\\ &\mathsf{FROM}\quad \mathtt{R},\ \mathtt{U},\ \mathtt{S},\ \mathtt{T}\\ &\mathsf{WHERE}\quad \mathtt{R}_\sigma = \mathtt{U}_\sigma\ \mathsf{AND}\ \mathtt{U}_\sigma = \mathtt{S}_\sigma\ \mathsf{AND}\ \mathtt{S}_\sigma = \mathtt{T}_\sigma,\end{aligned}$$

where R, U, S and T are relational tables, σ represents the filtering predicates (e.g., $\mathtt{R}.a < 10$, which returns tuples in R whose attribute a is smaller than 10) and the filtering operator can be $<$, $\leqslant$, $=$, $>$ and $\geqslant$. For a query, query optimizers find an efficient execution plan (usually expressed as a join tree) by enumerating feasible plans using algorithms such as dynamic programming [122]. The cost of a plan is calculated based on cardinality estimations of its sub-plans (e.g., $\mathtt{R}_\sigma \bowtie \mathtt{T}_\sigma$ and $(\mathtt{R}_\sigma \bowtie \mathtt{T}_\sigma) \bowtie \mathtt{U}_\sigma$ for the plan in Figure 5.2(a)). We focus on query-driven cardinality estimators and design LPCE to combat the large estimation errors for complex queries.

5.1.1 End-to-end Query Execution with LPCE

LPCE aims to reduce the end-to-end execution time of queries and consists of an initial estimation model LPCE-I and an estimation refinement model LPCE-R. LPCE

cooperates with the modern query engines (e.g., PostgreSQL) in manner of co-processing. LPCE serves the cardinality estimation in the query optimizer, which is processed by GPU. The other components in the query optimizer such as plan enumeration are processed by CPU. The query execution is processed by CPU. Figure 5.3 shows how LPCE works in the execution process of a query.

1. The query is sent to LPCE-I for initial cardinality estimations of all possible sub-plans when it is submitted;

2. Using the initial estimations, the query optimizer chooses a good execution plan using its plan search algorithm;

3. The chosen plan is executed and estimation refinement is triggered if the actual cardinality of some sub-plan differs significantly from the initial estimation;

4. For the feasible sub-plans of the remaining operators, LPCE-R is invoked to refine their cardinality estimations by exploiting the actual cardinalities of the executed sub-plans;

5. Based on the refined estimations, the query optimizer adjusts execution plan of the remaining operators for better efficiency.

The idea behind LPCE is simple: during the query execution process, the result tables of the executed sub-plans are available, which serve as inputs to the operators that remain to be executed and thus provide crucial information to refine their cardinality estimations (w.r.t. the initial estimations). LPCE is most beneficial for complex and long running queries, for which query-driven estimators can have large estimation errors and thus the initial execution plan may be far from optimal.

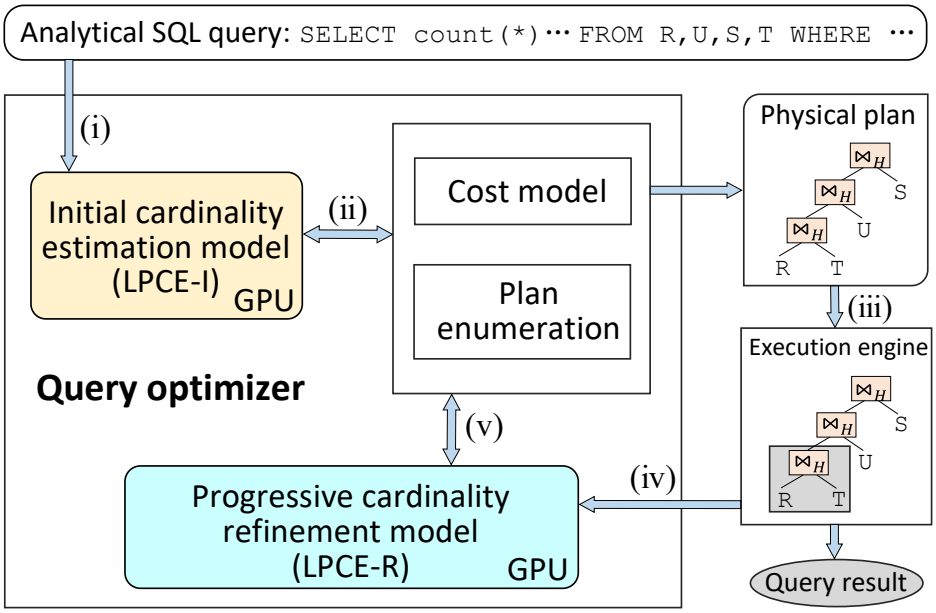


Figure 5.3: Using LPCE for query end-to-end execution.

5.1.2 Design Goals of LPCE

To reduce the end-to-end execution time of queries, LPCE needs to meet three key goals, i.e., *high accuracy, progressive refinement and fast inference*, which we elaborate as follows.

High Accuracy: Theoretically, query optimizers can find the optimal execution plan if the cardinality estimations and cost model are exact [122]. However, the errors of existing query-driven cardinality estimators can reach several orders of magnitude [122, 130]. To guide query optimizers to a good initial plan, LPCE-I should provide high accuracy for initial cardinality estimations. Accurate initial estimation also reduces the frequency of estimation refinement and query re-optimization, which incur extra overhead. For this goal, we design the node-wise loss function in Section 5.2.

Progressive refinement: LPCE-R should effectively reduce estimation error when more operators are executed such that re-optimization can find good execution plans.

For this purpose, the refinement model should fully utilize information of the executed sub-plans, especially their real cardinalities. To achieve this goal, we design a model structure with three modules for estimation refinement in Section 5.3, which explicitly considers the executed sub-plans.

Fast inference: Learning-based cardinality estimators are shown to significantly outperform traditional ones (e.g., histogram-based, sampling-based) in accuracy [109, 175, 212, 85, 82, 213]. However, a key concern of their applications in real database systems is the long inference time, which may outweigh their advantages in accuracy. Thus, both the initial estimation and progressive refinement models of LPCE should support fast inference. To achieve this goal, we use a light-weight model backbone and further compress it via knowledge distillation in Section 5.2.

5.2 Cardinality Estimation Model

In this part, we first introduce the generic pipeline for learning-based query-driven cardinality estimation models as background (Section 5.2.1), which we also follow in LPCE. Then we present the key designs of our initial cardinality estimation model (i.e., LPCE-I), including node-wise loss function, SRU-based light-weight model and knowledge distillation assisted model compression (Section 5.2.2∼5.2.4), which lead to high accuracy and fast inference.

5.2.1 Model Learning Pipeline

Given an execution plan, learning-based cardinality estimation models predict the cardinalities of the result tables (both indeterminate and final) [109, 175, 212, 82, 53]. As illustrated in Figure 5.4, training learning-based cardinality estimation models takes three steps, i.e., *sample collection, feature encoding and model learning*, which

we elaborate as follows.

Sample collection: Cardinality estimation is usually formulated as a regression problem, in which the execution plan is the input and the cardinalities are the output. Training samples can be collected from historical execution log. In the case of new database at "cold start", sample queries can be randomly generated according to the relational graph of the underlying dataset [109] and the execution plans can be obtained from database engines (e.g., via the `'EXPLAIN QUERY'` command in PostgreSQL). As shown in Figure 5.4, an execution plan is usually a tree, in which each node represents an operator (e.g., hash join, index scan and filter). Each node also contains detailed information about the query, such as filtering predicate (e.g., $t.kind\_id > 7$) and join condition (e.g., $\texttt{R}.a = \texttt{U}.a$). We can execute the plans and obtain the cardinality of each node using the `'Explain Analyze Query'` command or by adding counters.

Feature encoding: As machine learning models usually take vector input, the nodes in an execution plan are encoded into a feature vector during prepossessing. Following existing works [109, 175], we encode the *function, join condition, predicate* of each node as illustrated in Figure 5.5. Function is the logical operator at a node (e.g., join, scan) and we encode it as a one-hot vector with length $|P|$, where P is the set of all possible operators. Note that we consider function as logical operator rather than physical operator such as hash join and index scan. This is because cardinality estimation is done before physical operator chosen during plan generation. Join condition are the columns that are joined and we encode it as a two-hot vector with length $|C|$, where C is the set of all table columns for the dataset. For example, the join condition of node 3 is $[0, 0, 1, 0, 1, 0]$ in Figure 5.5, which indicates that the attribute value of column 3 needs to be equal to column 5 in the join. For nodes that do not conduct join, the join condition is a zero vector. *Predicate* is in the

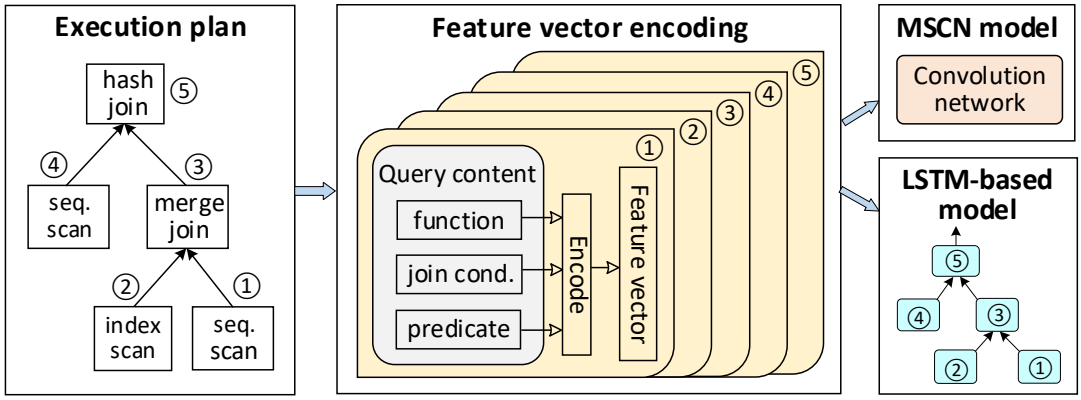


Figure 5.4: Workflow of learning-based estimation model.

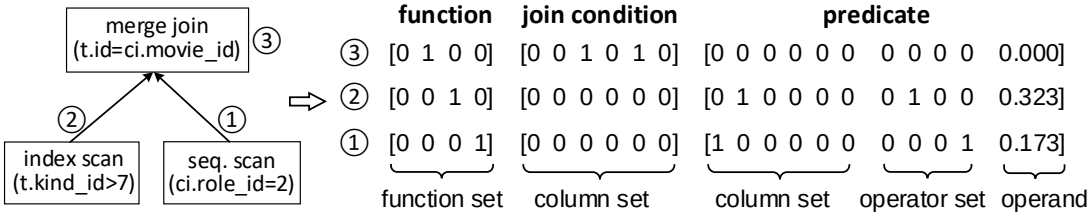


Figure 5.5: Example for feature encoding.

form of $[column_{id}, operator_{id}, operand]$ (e.g., $t.kind\_id > 7$), where $column_{id}$ and $operator_{id}$ use one-hot encoding while the value of *operand* is recorded as float after normalization. We concatenate the function, join condition, predicate vectors of a node to obtain a single feature vector and readers can refer to [109, 175] for more details about feature encoding.

Model learning: MSCN [109] and TLSTM [175] are two state-of-the-art models for query-driven cardinality estimation with different structures. MSCN stacks the feature vectors of all nodes in an execution plan and uses a multi-set convolution network [216] to map them to cardinality estimation. TLSTM processes an execution plan recursively following its tree structure using LSTM [86]. The embedding of the root node is treated as the query representation and used to predict cardinality. MSCN has large estimation errors as it does not utilize the structure of the execution plan while TLSTM suffers from the high computation complexity of LSTM. Our

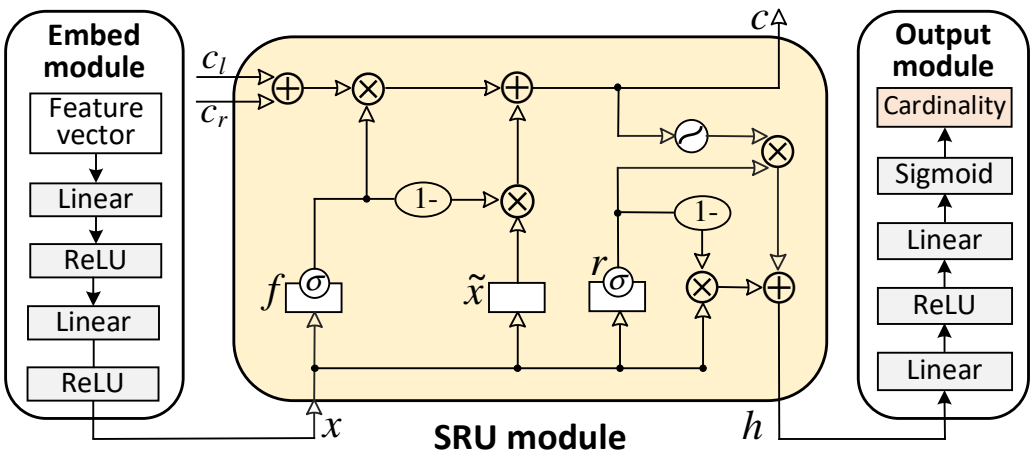


Figure 5.6: The structure of the SRU-based model in LPCE-I.

LPCE-I also processes an execution plan recursively using RNN as in TLSTM but designs are introduced to boost both accuracy and efficiency.

5.2.2 SRU-based Light Weight Model

Our SRU-based model is illustrated in Figure 5.6, which is used to process each node in an execution plan. The model consists of three modules, i.e., *embed module, SRU module and output module*. The embed module is used to map the sparse feature embedding of a node (introduced in Section 5.2.1) to dense embedding x. We use a two-layer fully-connected neural network with `ReLU` activation function as the embed module. The output module is used to map the node representation h (which encodes the sub-plan rooted at the node) to cardinality estimation, which is also two-layer fully-connected neural network. The final layer of the output module uses the `sigmoid` activation function to generate a float in [0,1], which is interpreted as the ratio of the estimated cardinality over the maximum cardinality observed in the train set.

For a node in an execution plan, the SRU module takes its query content embedding x, the encodings of its left and right child in the execution plan tree (i.e., c_l and c_r), as input to generate the node encoding c and node presentation h. The SRU

module is an RNN as node encoding c is passed to its parent node, which reuses the SRU module with the same parameter. The SRU module uses the simple recurrent unit (SRU) [121] as the model, which conducts computation following

$$\begin{aligned} \tilde{x} &= W_x x \\ f &= \rho(W_f x + b_f) \\ r &= \rho(W_r x + b_r) \\ c &= f \odot (c_l + c_r) + (1 - f) \odot \tilde{x} \\ h &= r \odot tanh(c) + (1 - r) \odot x \end{aligned} \tag{5.1}$$

where W_x, W_f and W_r are parameter matrices, b_f and b_r are bias vectors, $\odot$ denotes element-wise multiplication and ρ is the activation function `sigmoid`. f is the forget gate, which controls the weight of the children encodings (i.e., c_l and c_r) and projected node embedding (i.e., $\tilde{x}$) when generating encoding c for the current node. r is the reset gate, which uses the node embedding x and node encoding c to generate node representation for cardinality estimation.

We choose SRU as the RNN model in LPCE-I as it has higher computation efficiency than the LSTM in TLSTM. LSTM needs 8 matrix multiplications while SRU needs only 3. In addition, the 3 matrix multiplications (to compute $\tilde{x}$, f and r, respectively) in SRU can be parallelized while the matrix multiplications in LSTM have data dependencies. In the experiments, we show that our SRU-based model is 2.1x smaller in memory consumption, and 1.5x faster in inference speed compared with TLSTM when running on GPU. We also show that changing from LSTM to SRU has negligible loss in estimation accuracy.

5.2.3 Node-wise Loss Function

Both MSCN and TLSTM use the mean q-error as the loss function, which is defined as

$$\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n} q_i \quad \text{with } q_i = \frac{\max(c_i, \tilde{c}_i)}{\min(c_i, \tilde{c}_i)}, \tag{5.2}$$

where c_i and $\tilde{c}_i$ are the real and model estimated cardinalities for the final result table of the i^{th} execution plan, and the train set contains n plans. q_i is the q-error of the i^{th} plan, which measures the estimation accuracy and lower value indicates better estimate (note that $q_i \geqslant 1$). As Equation (5.2) only considers estimation errors for the final result of each query, we call it the *query-wise loss function*. In LPCE-I, we use the *node-wise loss function* defined as

$$\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{m_i} q_{ij} \quad \text{with } q_{ij} = \frac{\max(c_{ij}, \tilde{c}_{ij})}{\min(c_{ij}, \tilde{c}_{ij})}, \tag{5.3}$$

where c_{ij} and $\tilde{c}_{ij}$ are the real and model estimated cardinality for the j^{th} node in the i^{th} execution plan, and m_i is the number of nodes in the i^{th} execution plan. Different from the query-wise loss function, the node-wise loss function considers the q-error of all nodes in each execution plan.

We observe that the node-wise loss function significantly improves estimation accuracy and the reasons are two-fold. First, it is a kind of data augmentation that enlarges the train set. For example, for a single execution plan $(A \bowtie B) \bowtie (C \bowtie D)$ in the train set, the node-wise loss function actually uses the estimation error of three execution plans, i.e., $(A \bowtie B)$, $(C \bowtie D)$ and $(A \bowtie B) \bowtie (C \bowtie D)$. As complex queries contain many internal nodes, the data augmentation effect is significant. Second, it allows supervision for every node in the execution plan. TLSTM (and our LPCE-I) uses RNN to embed an execution plan from leaf to root along its tree structure and the query-wise loss function only provides supervision for the root node,

which means that the gradients need to back-propagate in time to reach the internal nodes. By providing direct supervision signal for the internal nodes, the node-wise loss function produces more informative embedding for the internal nodes, which yields more accurate representation and thus cardinality estimation for the entire query. We observe that the gain of the node-wise loss function is significant for complex execution plans with a deep tree structure (Section 5.5).

5.2.4 Knowledge Distillation for Compression

The model structure parameters of LPCE-I (e.g., the size of the embedding vectors and number of hidden units in the neural networks) control the complexity and accuracy of the model. Although using a small model provides fast inference, we found directly training the small model yields poor accuracy. This is because smaller models have weaker ability to learn and generalize. To train small model with high accuracy, we use *knowledge distillation* [69], which uses a large (thus accurate) *teacher model* to guide the learning of the small *student model*. The idea is that the student model can learn useful knowledge by matching the output of the teacher model as shown in Figure 5.7, which is easier to fit than training data as it is produced by the teacher model whose structure is the same as the student model.

To conduct knowledge distillation, we first train a teacher model with high complexity and accuracy. Then, we train the student model using the following hint loss

$$\mathcal{L}_{\text{hint}} = \frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{m_i} \|x_{ij}^t - p_e(x_{ij}^s)\|_1 + \|h_{ij}^t - p_s(h_{ij}^s)\|_1. \tag{5.4}$$

For the j^{th} operator in the i^{th} execution plan, x_{ij}^t and x_{ij}^s are the output of the embed module of the teacher model and student model, respectively. Similarly, h_{ij}^t and h_{ij}^s are the node representation produced by SRU module of the teacher model and student model. $p_e(.)$ and $p_s(.)$ are two single-layer neural networks that are used

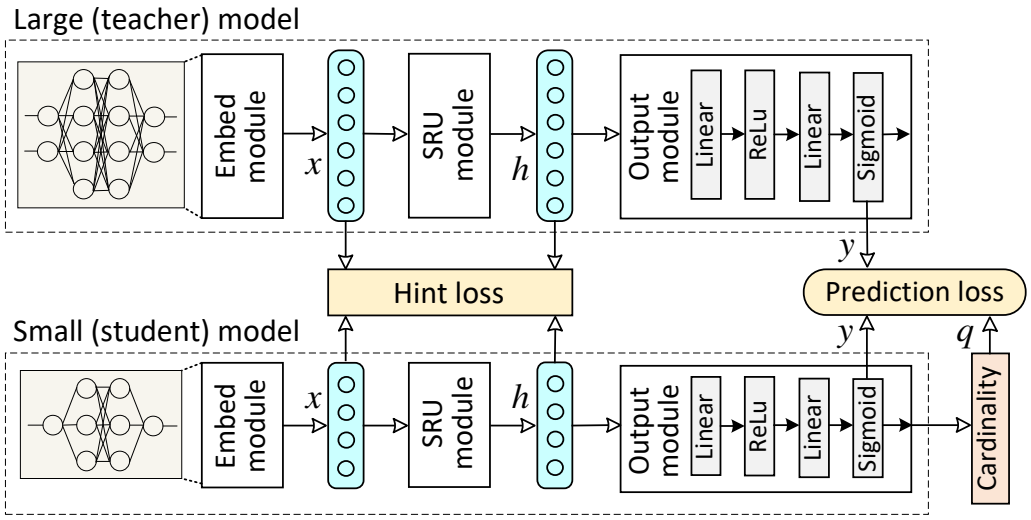


Figure 5.7: Model compression via knowledge distillation.

to adjust the outputs of the student model to the same size as the teacher model. After that, we further calibrate the student model using the following prediction loss

$$\mathcal{L}_{\text{predict}} = \frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{m_i} \alpha q_{ij}^{s} + (1-\alpha)\left|y_{ij}^{t} - y_{ij}^{s}\right|, \tag{5.5}$$

where q_{ij}^{s} is the q-error for the cardinality estimation of the student model, y_{ij}^{t} and y_{ij}^{s} are the logit before the `sigmoid` activation function in the output module of the teacher and student model. We train the student model to fit the logit of the teacher model as it has a direct impact on cardinality estimation. α is a weight used to balance the two loss terms and usually set as 0.5. In the experiments, we show that knowledge distillation can compress the LPCE-I model by more than 10x and speed up inference without degrading accuracy. The model inference is faster on GPU due to the smaller model size.

5.3 Cardinality Refinement Model

Query-driven estimator produces cardinality prediction at large errors for queries that model has not learned. We propose to query re-optimization with cardinality

refinement to combat the large errors and adjust the execution plan to be better one. In this part, we present the design of LPCE-R, our model for progressive cardinality estimation refinement.

5.3.1 The Model Structure of LPCE-R

As illustrated in Figure 5.8, query execution in database systems is conducted from leaf to root along the execution plan tree. At some points in time, some nodes in the query plan (shadowed in Figure 5.8) are finished while some remain to be executed. The idea of LPCE-R is to use the information of the finished nodes to refine cardinality estimations for the remaining nodes as the cardinalities of the remaining nodes depend on the intermediate result tables of the finished nodes. Thus, LPCE-R needs to solve three problems: (i) what information should we extract from the executed sub-plan, (ii) how to effectively exploit the information for cardinality refinement and (iii) how to efficiently refine the cardinality estimations in the query execution process.

To address the problems above, LPCE-R adopts a hybrid structure with three modules as illustrated in Figure 5.8. All three modules adopt the same structure as LPCE-I in Section 5.2.2 but are trained differently (will be discussed in Section 5.3.2). Cardinality module and content module are used to extract different information from the executed sub-plan. A connect layer merges the information from cardinality module and content module as input for refine module, which refines the cardinality estimation for the remaining nodes. In the following, we introduce the design of each of these components.

Information extraction with two modules: Two kinds of information can and should be extracted from the executed sub-plan, i.e., *real cardinality* and *query content*. For example, if the cardinality of a node is estimated as 10 originally but the real cardinality is found to be 1000 after finishing it. Noticing this 100x underesti-

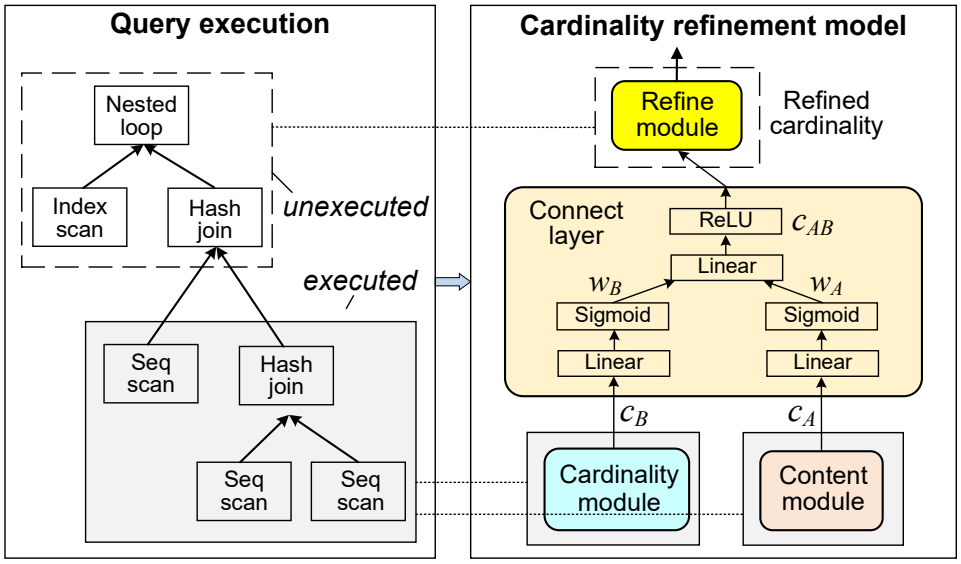


Figure 5.8: An overview of LPCE-R for estimation refinement.

mation is crucial for refining the estimations for the remaining nodes as the results of these nodes depend on the finished nodes. In addition, the semantics of the executed sub-plan (i.e., query content), such as the type of the operators, joined columns and filtering predicates, are also important for estimating the cardinalities of the remaining nodes. For example, a filtering predicate in the executed sub-plan (e.g., $R.a < 10$) could have a direct impact on the remaining join operators (e.g., $R \bowtie U$ with $R.a = U.a$). Therefore, cardinality module is used to extract the real cardinality of the executed sub-plan and adopts the structure of LPCE-I. Content module is used to extract the query content and its structure is also exactly the same as LPCE-I. One can append the real cardinalities of its two children to the query content feature encoding of an executed node, which is used as input for the single module. However, in the experiments, we observed using both cardinality and content module yields significantly higher accuracy than using only one module, which suggests that both real cardinality and query content are important for estimation refinement.

Learned information merge: Both cardinality and content module produce an

embedding (i.e., c_A and c_B), which encodes the executed sub-plan. As shown in Figure 5.8, we train a connect layer to merge the two embedding as input for refine module. The connect layer uses a single-layer neural network with `sigmoid` activation function to learn the merge weights for c_A and c_B, respectively. The weighted combination of c_A and c_B is processed by a single-layer neural network with ReLU activation function. Specifically, the connect layer conducts the following computation

$$\begin{aligned} w_A &= \rho(W_A c_A + b_A) \\ w_B &= \rho(W_B c_B + b_B) \\ c_{AB} &= \mathrm{ReLU}\big(W_{AB}(w_A \odot c_A + W_B \odot c_B) + b_{AB}\big), \end{aligned} \tag{5.6}$$

where $\rho(\cdot)$ is the `sigmoid` function and $\odot$ denotes element-wise multiplication. W_A, W_B and W_{AB} are parameter matrices. We use a learned connect layer because we observed empirically that the contributions of real cardinality (i.e., c_B) and query content (i.e., c_A) to accuracy vary in different scenarios. For example, when the number of remaining operators is small, using only real cardinality already provides good estimation accuracy. However, the query content is important when there are many remaining operators due to more dependencies on the semantics of the finished sub-plan. The connect layer can learn to adjust the weights of real cardinality and query content according to the scenario.

Efficient progressive refinement: Refine module is used to refine the cardinality estimations of the remaining operators and its structure is exactly the same as LPCE-I. The embedding of the executed sub-plan (i.e., c_{AB}) is fed as input to the SRU of refine module, which replaces both or either of c_l and c_r (see Figure 5.6) according to the execution situation. When an operator finishes, both cardinality and content module update their embedding according to the previously executed sub-plan (which has already been processed by the modules) and the just finished

operator. Using the updated embedding, refine module refines the estimations for all remaining operators. Progressive refinement is efficient as cardinality and content module do not need to process the entire finished sub-plan from scratch. Instead, the embedding is updated incrementally with each finished operator.

5.3.2 Training Procedures

As illustrated in Figure 5.9, the training of LPCE-R consists of two stages, i.e., *pre-train* and *adjustment*. In the pre-train stage, content module is trained in the same way as LPCE-I and refine module shares the same parameters as content module. For cardinality module, we concatenate the feature encoding (for the query content) of each operator with the real cardinalities of its two children as input, and train the module to minimize the node-wise loss function in Equation (5.3). Note that for the leaf nodes in an execution plan, their real cardinalities are the number of tuples in the considered attributes.

In the adjustment stage, we froze cardinality and content module, and fine-tune refine module. In this case, an execution plan with m operators provides m-1 training samples for refine module. Specifically, when each operator finishes, content and cardinality module are used to obtain the embedding of the executed sub-plan, and refine module is trained to predict the cardinalities of the remaining operators. The loss function is also the node-wise loss function in Equation (5.3). We provide such an example in the lower part of Figure 5.9, in which operators 4, 5, 6 are executed and the cardinalities of operators 0 and 2 need to be re-estimated.

5.4 Integrating LPCE into PostgreSQL

The query optimizer of PostgreSQL uses dynamic programming to enumerate all possible execution plans. For a query that joins n relation tables, enumeration starts at level-1 for the base tables (e.g., $\mathtt{R}_{\sigma}$, along with possible filtering predicates) and

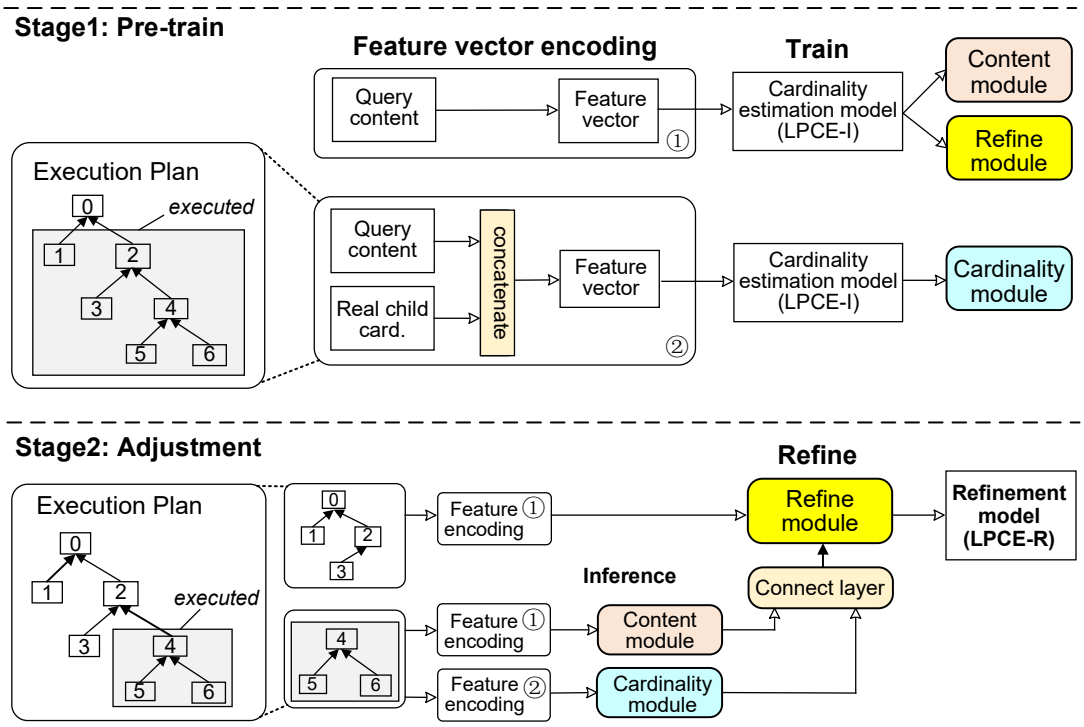


Figure 5.9: The training workflow of LPCE-R.

ends at level-n for the entire query (e.g., $\mathtt{R}_\sigma \bowtie \mathtt{T}_\sigma \bowtie \mathtt{U}_\sigma$). To find the best execution plan for a sub-query at level-i, the optimizer decomposes it into smaller sub-queries at lower levels (called children) and requires cardinality estimations of all children. For example, optimizing $\mathtt{R}_\sigma \bowtie \mathtt{T}_\sigma \bowtie \mathtt{U}_\sigma$ requires cardinality estimations of $\mathtt{R}_\sigma$, $\mathtt{T}_\sigma$, $\mathtt{U}_\sigma$, $\mathtt{R}_\sigma \bowtie \mathtt{T}_\sigma$, $\mathtt{T}_\sigma \bowtie \mathtt{U}_\sigma$ and $\mathtt{R}_\sigma \bowtie \mathtt{U}_\sigma$. Thus, a query that joins n relations requires up to $2^n - 1$ cardinality estimations (when all relations can join with each other).

Integrating LPCE-I: We replace the histogram-based cardinality estimator of PostgreSQL with LPCE-I. Following [175], a *memory pool* is used to store the cardinality estimations and execution costs of sub-queries. We batch the inference for all sub-queries on the same level as they have the same number of inputs and the feature vectors of a sub-query are small. For PostgreSQL, query execution time can be decomposed into planning time T_P and execution time T_E as its histogram-based cardinality estimator has negligible overhead. For learning-based estimators, end-to-

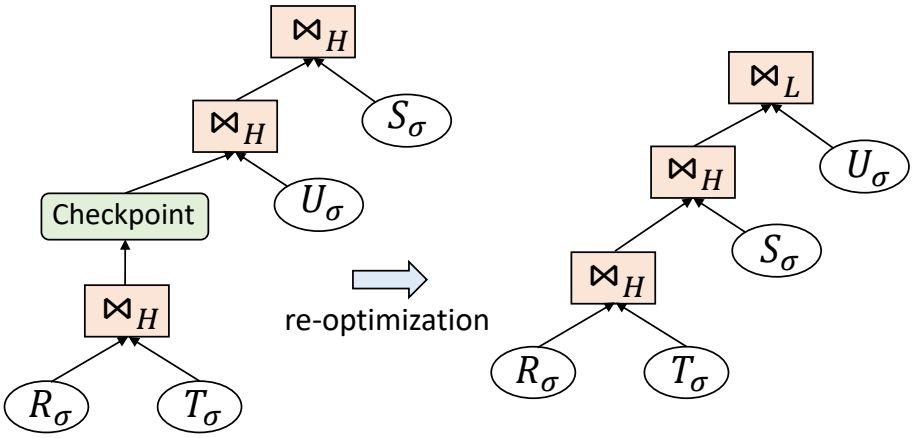


Figure 5.10: Example of query re-optimization with LPCE-R.

end query execution time can be expressed as

$$T_{end} = T_P + T_I + T_E, \tag{5.7}$$

where T_I is the model inference time. For queries with a short execution time T_E, T_I can easily dominate T_{end}, in which case learning-based estimators may have longer end-to-end time than PostgreSQL. In the experiments in Section 5.5, we show that LPCE-I outperforms PostgreSQL in most cases due to its short inference time.

Integrating LPCE-R: As PostgreSQL does not provide native support for query re-optimization, we follow the *checkpoint scheme* in [135]. As shown in Figure 5.10, we place checkpoints at all join operators in the execution plan, which assert whether the real cardinality of the operator is significantly different from the (initially) estimated cardinality. Join implementations in PostgreSQL include hash join, merge join, and nested loop join. Hash join and merge join are blocking operations, which need to terminate before subsequent operations start. Checkpoints are placed at hash table building stage for hash join, and before sorting starts for merge join. Nested loop join is pipelined with subsequent operations in PostgreSQL, and thus we block nested loop join and place checkpoint at the outer side of the nested loop.

We use the q-error to measure the difference between the actual cardinalities

and initially estimated cardinalities, and continue executing the original plan if the q-error is below a threshold (empirically set as 50). Otherwise, re-optimization is triggered and the intermediate results are packaged into a *materialized view*, which can be used by subsequent operators. During re-optimization, LPCE-R is invoked to provide refined cardinality estimations, and the optimizer searches the optimal execution plan among those that continues from the materialized view and those that restarts query processing from scratch. The end-to-end execution time of a query that triggers re-optimization can be expressed as

$$T_{end} = T_P + T_I + T_R + T_E, \tag{5.8}$$

where T_R is the re-optimization time, which includes the plan search and model inference time during re-optimization. Therefore, re-optimization should be triggered when its execution time reduction outweighs T_R. Currently, we use a simple threshold, which may invoke unnecessary re-optimizations or miss re-optimization opportunities. The experiments in Section 5.5 show that re-optimization with this simple rule already brings benefits and we leave designing more sophisticated policies to trigger re-optimization as future work. From Equation (5.8), we also observe that the short inference time of LPCE-R is crucial for enjoying the benefits of re-optimization.

5.5 Experimental Evaluation

In this section, we present our empirical studies. Section 5.5.1 introduces the experiment settings. Section 5.5.2 evaluates the end-to-end query execution time of LPCE and compares with state-of-the-art solutions. Section 5.5.3 validates the effectiveness of the key designs (e.g., node-wise loss function, model compression) of LPCE.

5.5.1 Experiment Settings

Workloads: We used the IMDB dataset [122] for the experiments, which contains 22 tables recording facts (e.g., actors, directors and companies) about over 2.1M movies. IMDB is widely utilized in related researches [175, 109, 213, 212, 118, 123] as a challenging benchmark for cardinality estimation due to its non-uniform data distributions and complex correlations among the tables. We generate the training and testing queries according to the relational graph for the tables in IMDB following [109]. The training set contains 10,000 sample queries with 6-8 joins, and 10% of these queries are randomly selected as validation. We used two query sets for performance testing: (1) *Join-six*, 500 queries with 6 joins; and (2) *Join-eight*, 500 queries with 8 joins. As shown in Figure 5.11, we select the test queries such that their end-to-end execution time on PostgreSQL spreads over a wide range (i.e., from 1s to 1,500s).

Baselines: We compared our LPCE with the following state-of-the-art learning-based cardinality estimators.

- 2 query-driven estimators, i.e., MSCN [109] and TLSTM [175]. MSCN uses a multi-set convolution network to map the query feature vector to cardinality value while TLSTM uses an LSTM (a kind of recurrent neural network) to estimate the cardinality of a query according to its execution plan.
- 2 data-driven estimators, i.e., DeepDB [85] and NeuroCard [212]. DeepDB adopts the *sum product networks* (SPN) to capture the joint distribution of the relations and assumes conditional independence across the relations. NeuroCard removes the independence assumption and builds a single deep autoregressive (AR) model over all relations.
- 1 hybrid estimator, UAE [200], which learns from data in an unsupervised manner

and query samples in a supervised manner.

Our methods include LPCE-I, which only conducts cardinality estimation before query execution (the same as existing learning-based estimators), and LPCE-R, which uses LPCE-I for initial estimation and may trigger query re-optimization and progressive estimation refinement if the errors are found to be large. We use end-to-end query execution time as the main performance metric.

Implementation details: For MSCN[1], DeepDB[2], TLSTM[3], and NeuroCard[4], we used their open-source implementations and adopted the hyper-parameters recommended by their authors. For UAE, we obtained the implementation from its authors. Since MSCN and DeepDB do not support range queries on categorical string columns, we encode these columns into integers using dictionary encoding. As described in Section 5.4, we replace the histogram-based cardinality estimator of PostgreSQL with the learning-based estimators for performance evaluation.

For our LPCE-I, the number of hidden units for the embedding module, SRU module and output module are 64, 196 and 1024, respectively. LPCE-I is compressed from a large model via knowledge distillation, for which the number of hidden units for the embedding module, SRU module and output module are 256, 1024 and 1024, respectively. For LPCE-R, cardinality refinement is triggered when the q-error between the actual cardinality of an intermediate result table and the initial estimation is above 50. The models are trained with a batch size of 50 and the Adam optimizer.

Our implementations were based on Python (v3.7.5), PyTorch (v1.5.1), and PostgreSQL (v13.0). All experiments were conducted on a machine with Intel(R) Xeon(R) Gold 5122 CPU, 512 GB DRAM, and Nvidia Tesla V100s GPU. GPU is

[1]https://github.com/andreaskipf/learnedcardinalities/

[2]https://github.com/DataManagementLab/deepdb-public

[3]https://github.com/greatji/Learning-based-cost-estimator

[4]https://github.com/neurocard/neurocard

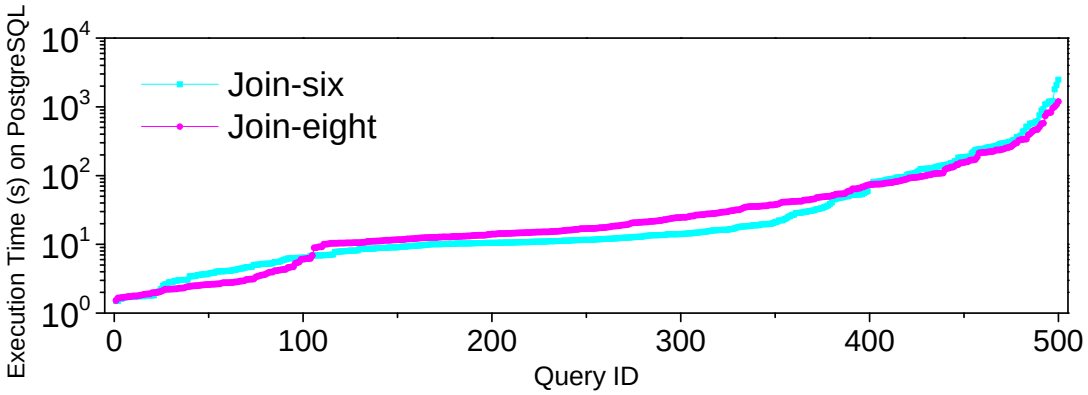


Figure 5.11: Execution time of the test queries on PostgreSQL. Queries are ordered in the ascending order of the execution time.

used as default processor for the learning-based estimators. Our method LPCE, and baselines MSCN, TLSTM, NeuroCard and UAE are processed with GPU by default. DeepDB does not support running on GPU, and is processed with CPU by default. Our source code and query sets are open-source on github[5].

5.5.2 End-to-End Query Execution Time

For a query, we define the *execution time reduction* (reduction for short) of a learning-based estimator w.r.t. PostgreSQL as

$$R = \frac{T_{Postgres} - T_{Learn}}{T_{Postgres}}, \tag{5.9}$$

in which $T_{Postgres}$ and T_{Learn} are the end-to-end query execution time of PostgreSQL and the learning-based estimator, respectively. A negative reduction means that the learning-based estimator has longer execution time than PostgreSQL, and a large reduction indicates a significant speedup (e.g., a reduction of 99% means a 100x speedup over PostgreSQL). In Table 5.2, we report some percentiles of the reduction for the learning-based methods, in which the 5th percentile and 95th percentile correspond to worst and best cases. We also plot the end-to-end execution time for the

[5]https://github.com/Eilowangfang/LPCE

Table 5.2: Percentiles of execution time reduction compared to PostgreSQL (large the better, best marked in bold).

Join-six	5th	25th	50th	75th	95th
DeepDB	-226%	7.87%	56.4%	88.6%	98.2%
NeuroCard	-115%	11.9%	45.9%	71.8%	97.4%
UAE	-116%	44.7%	64.4%	89.7%	98.9%
MSCN	-240%	37.7%	70.8%	91.2%	99.4%
TLSTM	-118%	40.3%	72.9%	92.3%	98.0%
LPCE-I	-11.17%	69.3%	85.5%	94.2%	99.6%
LPCE-R	**-3.00%**	**71.4%**	**86.9%**	**96.6%**	**99.7%**
Join-eight	5th	25th	50th	75th	95th
DeepDB	-445%	-22.6%	22.9%	71.7%	95.6%
NeuroCard	-370%	-14.3%	29.6%	74.5%	95.7%
UAE	-360%	-8.85%	35.2%	75.8%	95.8%
MSCN	-975%	22.1%	72.4%	82.4%	98.1%
TLSTM	-389%	34.3%	75.8%	92.1%	98.6%
LPCE-I	-28.7%	73.1%	86.9%	95.9%	99.5%
LPCE-R	**-9.52%**	**74.7%**	**87.0%**	**96.3%**	**99.4%**

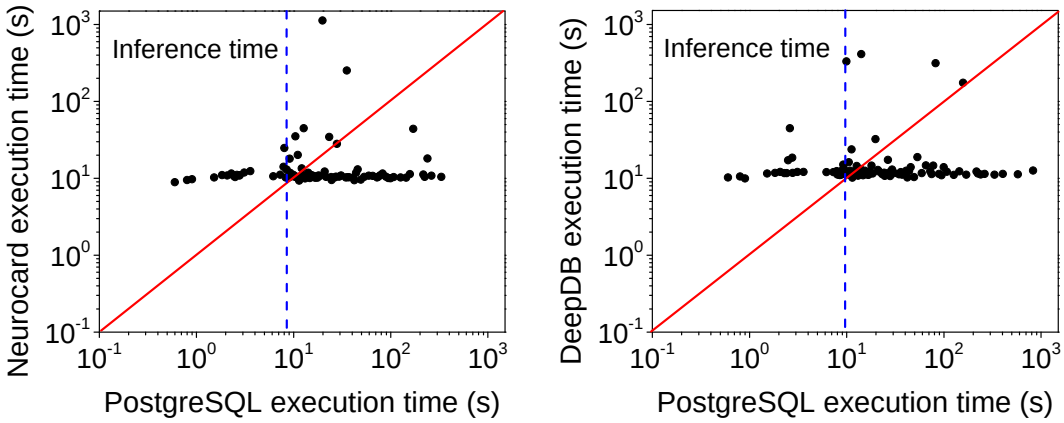
learning-based estimators as scatter plot in Figure 5.12. A point on the left of the diagonal line indicates the learning-based estimator has longer execution time than PostgreSQL, and the dotted line indicates the model inference time. The scatter plots for the *Join-six* queries resemble Figure 5.12. We make 3 observations from Table 5.2 and Figure 5.12.

1. Short model inference time is crucial: Table 5.2 shows that data-driven and hybrid estimators (i.e., DeepDB, NeuroCard and UAE) have poor performance at 5th percentile (also 25th percentile for *Join-eight*), and can significantly prolong the execution time of PostgreSQL for some queries. Figure 5.12 shows that this can be explained by their long inference time, which is several to tens of seconds and they cannot speed up queries whose execution time on PostgreSQL is shorter than their inference time (i.e., queries on the left of the dotted line in Figure 5.12). The

cost of model inference also explains why data-driven and hybrid estimators have worse 5th percentile on *Join-eight* than *Join-six*—a query needs up to 127 and 511 cardinality estimations for *Join-six* and *Join-eight*, respectively. In contrast, the query-driven estimators (i.e., MSCN, LSTM and LPCE-I) generally performs better at 5th percentile because of their short inference time, which is typically below 1 second (including the time of parsing query and encoding feature).

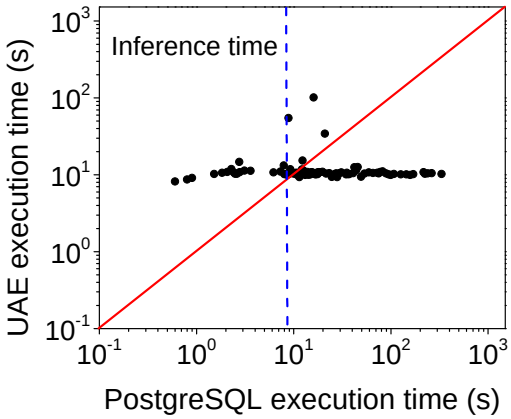
2. High estimation accuracy matters: Table 5.2 also shows that the learning-based estimators speed up PostgreSQL in most cases. For example, LPCE achieves a reduction of 99.7% at 95th percentile for *Join-six* queries, which corresponds to a speedup of about 330x. We also observed that learning-based estimators can reduce the execution time from around 1,000s for PostgreSQL to several seconds. This is because the histogram-based estimator of PostgreSQL has poor accuracy and thus leads to poor execution plans. Among the same type of learning-based estimators, estimation accuracy is also important to performance. Although DeepDB, NeuroCard and UAE have similar inference time according to Figure 5.12, UAE has the best reduction in Table 5.2 because it jointly utilizes data-based training and query-based training to achieve high accuracy. In contrast, DeepDB has the worst reduction in Table 5.2 due to the poor accuracy caused by its independence assumptions. In the query-driven estimators, MSCN has significantly shorter inference time than LSTM and LPCE-I, but its reduction is much worse than LSTM and LPCE-I because of poor estimation accuracy.

3. LPCE balances inference time and estimation accuracy: Both Table 5.2 and Figure 5.12 show that LPCE consistently outperforms the baselines across different query sets and percentiles. The 5th percentiles of LPCE are small (below 10%), which resolves the popular concern that a learning-based estimator may significantly degrades performance for some queries. For more optimistic cases (e.g., 75th or

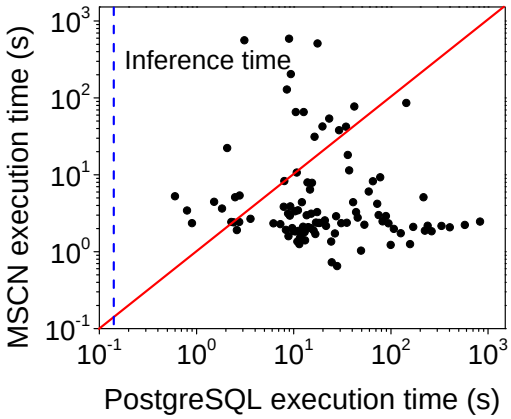


(a) NeuroCard

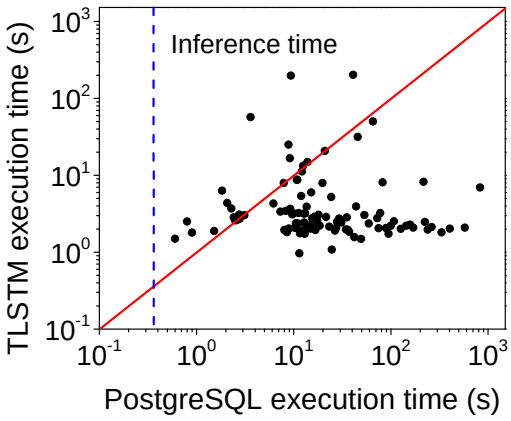
(b) DeepDB



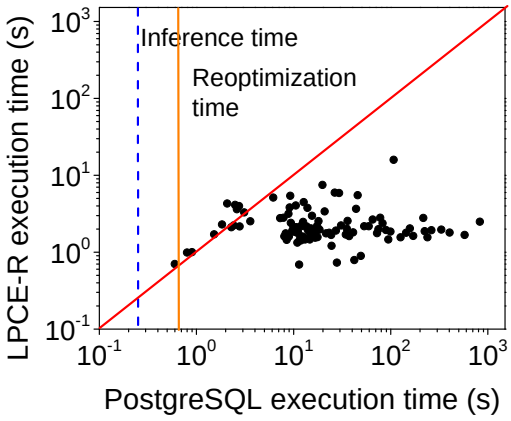
(c) UAE



(d) MSCN

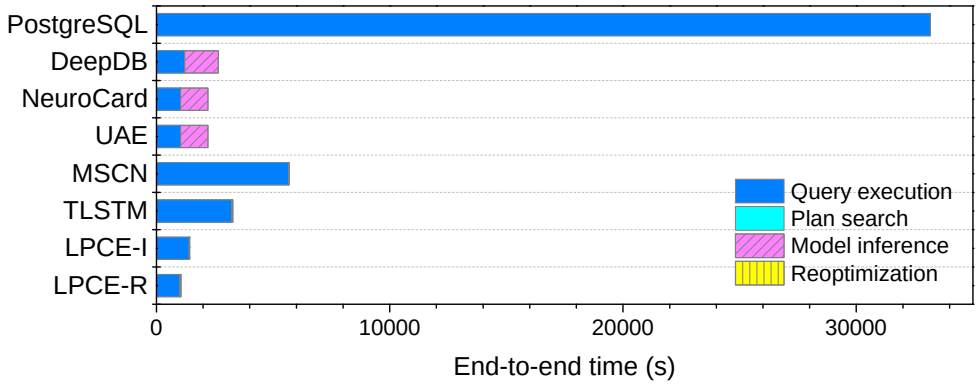


(e) TLSTM

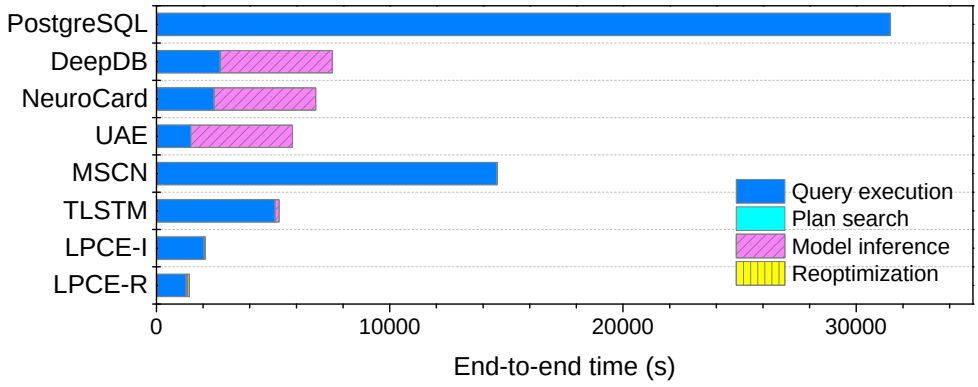


(f) LPCE-R

Figure 5.12: End-to-end execution time of the learning-based estimators and PostgreSQL for *Join-eight* queries.



(a) *Join-six*



(b) *Join-eight*

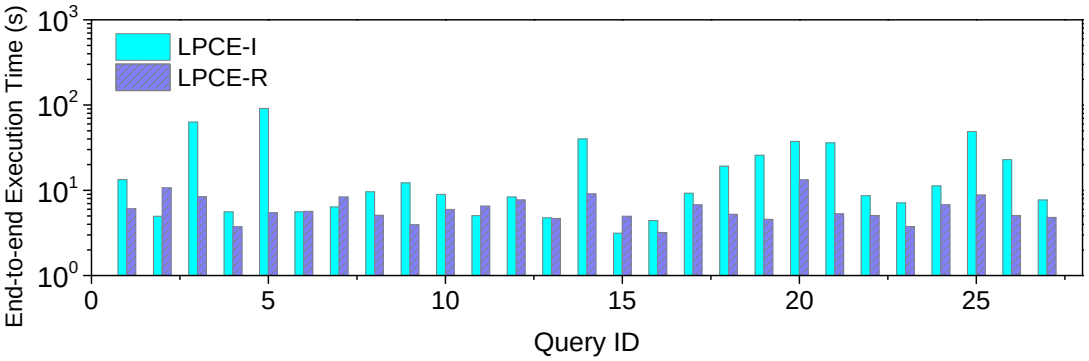
Figure 5.13: Decomposition of end-to-end query execution time, statistics are aggregated over the 500 test queries.

95th percentile), LPCE achieves larger or comparable speedup over PostgreSQL when compared with other learning-based estimators. LPCE has good performance because it balances inference time and estimation accuracy, which we show in Figure 5.13 by decomposing the end-to-end execution time of the queries. The results show that for methods with poor cardinality estimation accuracy (e.g., PostgreSQL and MSCN), the query execution time is long as bad execution plans are used. For methods with a long inference time (e.g., UAE and NeuroCard), although query execution time is short, model inference could dominate end-to-end query execution. LPCE-R benefits from using LPCE-I for an efficient and accurate initial estimation, and progressively refining the estimations during re-optimization. Although the re-optimization time of

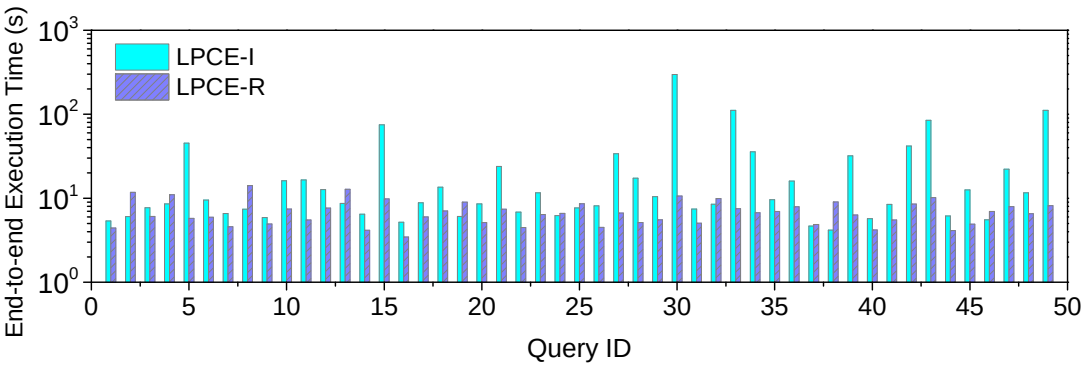
LPCE-R is long as reported in Figure 5.12, Figure 5.13 shows that the re-optimization time is negligible in end-to-end time. This is because re-optimization time is only triggered for a few queries with large estimation errors, which we will show shortly.

Advantages of progressive estimation refinement: Among the 500 test queries, 49 and 27 queries incur progressively estimation refinement for *Join-six* and *Join-eight*, respectively. To show the gain of progressive refinement, we compare the execution time of these queries with LPCE-I and LPCE-R in Figure 5.14. The results show that LPCE-R speeds up most of the queries and the speedup is the most significant for queries whose execution time is long with LPCE-I. This is because LPCE-R can correct the significant estimation errors of LPCE-I, which leads to bad execution plans. For a few queries, LPCE-R has longer end-to-end execution time than LPCE-I but the performance degradation is small. This is because LPCE-R still has large errors on some difficult queries and re-optimization comes with overhead. In Figure 5.15, we report the time decomposition for the re-optimized queries. The results show that LPCE-R significantly reduces the overall execution time of the re-optimized queries and the re-optimization overhead is small. The re-optimization time includes model inference and materialization of intermediate results by checkpoint. We observed that materialization cost typically dominates the re-optimization time and is determined by the size of the intermediate results.

To explain the performance gain of LPCE-R over LPCE-I, Figure 5.16 shows how the estimation errors of LPCE-R change in the query execution process. The query plan has 14 operators and 18 operators for *Join-six* and *Join-eight*, respectively. The results in Figure 5.16 shows that LPCE-R effectively reduces the estimation error in the query execution process. For example, when the number of executed operators increases from 3, 6, 9 to 12 on *Join-six*, LPCE-R gradually reduces the mean q-error from 33.5, 22.7, 17.4 to 10.3. One interesting observation is that there are some



(a) *Join-six*



(b) *Join-eight*

Figure 5.14: Re-optimization of LPCE-R.

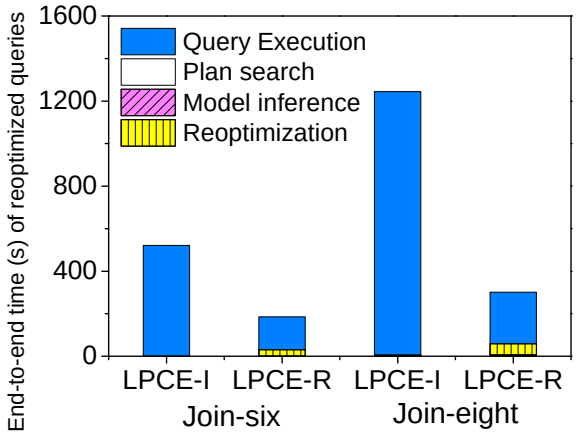
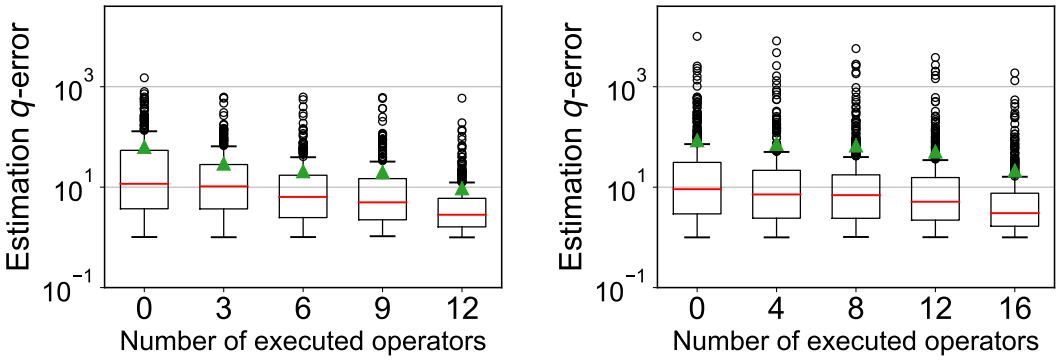


Figure 5.15: Time decomposition for the re-optimized queries.



(a) *Join-six* (b) *Join-eight*

Figure 5.16: The change of the mean q-error for LPCE-R in the query execution process.

difficult queries whose estimation errors are large even when many operators are executed. We conjecture that this is because these queries do not having matching samples in the training set and thus the model cannot generalize to them.

Limitations of LPCE: Although LPCE-R outperforms the baselines in the previous experiments, there can be cases in which the baselines have short end-to-end execution time than LPCE. One such case is when the queries have a small number of joins. For a query joining n relations, plan search in PostgreSQL needs to estimate up to 2^n-1 cardinalities. Thus, when n is small, the high inference overhead of data-driven and hybrid estimators becomes less significant. In addition, data-driven and hybrid estimators typically have better estimation accuracy than query-driven estimators. To provide such an example, we report the end-to-end execution time of 500 queries with 3 joins in Figure 5.17. The results show that NeuroCard and UAE perform the best among the estimators, and the primary reason is that model inference overhead is lower compared with more complex queries.

LPCE on varying datasets and workloads: We compare the LPCE with baseline estimators on more datasets and workloads. We use the standard dataset TPC-H [182] with scale factor 5 (GB) and two query sets: *TPC-ben* has 22 queries from the

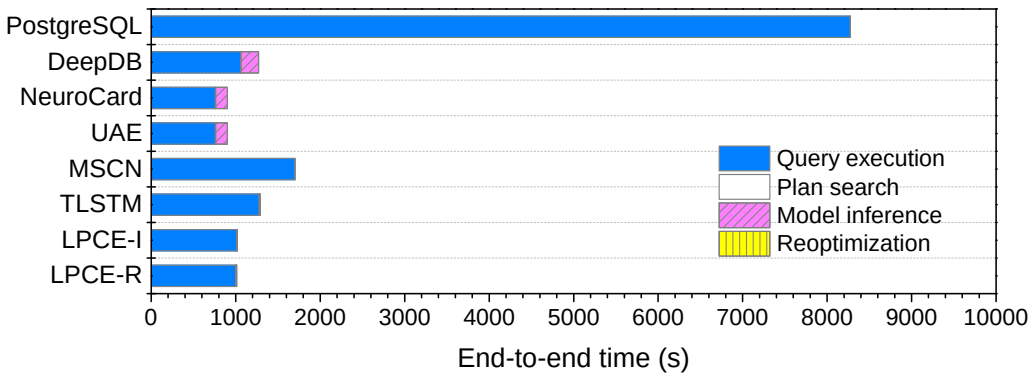


Figure 5.17: End-to-end execution time for 3-join queries.

standard benchmark; *TPC-join-six* has 300 queries with 6 joins, which we generate according to the relational graph for the tables in TPC-H. We use 10K samples to train the LPCE. We also follow the recent works [195, 106, 202] to generate *Synthetic* dataset with multiple tables. Specifically, the dataset has 10 tables and each table has 1M data tuples. The tables are correlated with primary key and foreign key join pairs. The query set *Synthetic-join-eight* has 300 queries with 8 joins, and the training set has 10K samples with 8 joins.

Figure 5.18(a) shows LPCE has slight improvement on *TPC-ben* compared to PostgreSQL. Though LPCE designed for select-project-join (SPJ) queries, we follow [163, 211] to decompose the query with sub-queries into SPJ blocks and optimize them in manner of block-by-block. Note that DeepDB and MSCN do not support complex predicates with string values such as 'like', and thus not included for comparison. The results show that the estimators NeuroCard, UAE and TLSTM have similar performance with LPCE. The limited speedup is because that cardinality estimation is not hard on *TPC-ben*, and PostgreSQL has small estimation errors on the queries. We conjecture there are two reasons. First, TPC-H are generated using the simplifying assumptions of uniformity and independence, which follows the same assumptions that query optimizer of PostgreSQL makes [122]. Second, queries of *TPC-ben* has small number of joins and filters, thus easy tasks for cardinality esti-

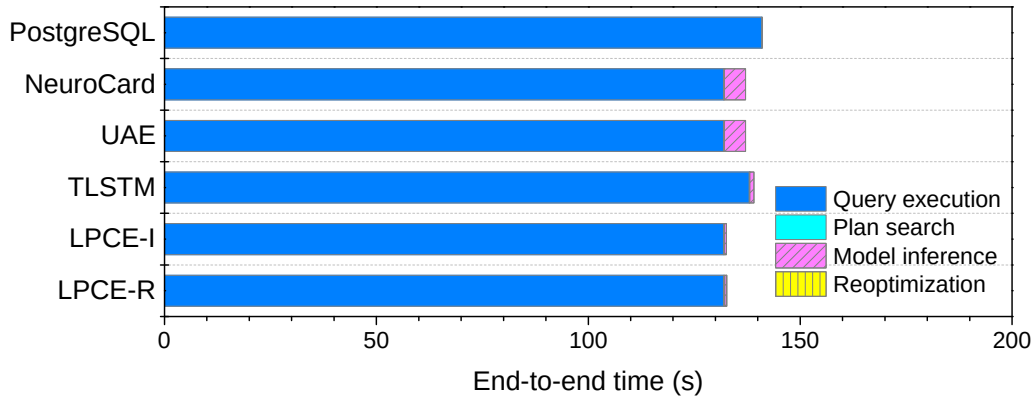
Table 5.3: Cardinality estimation q-error on PostgreSQL for *TPC-ben* and *Join-eight*.

q-error	25th	50th	75th	90th	95th	99th	max
TPC-ben	1.00	1.01	1.90	12.9	626	4886	12472
Join-eight	16.3	463	9302	153174	752717	5165938	32179314

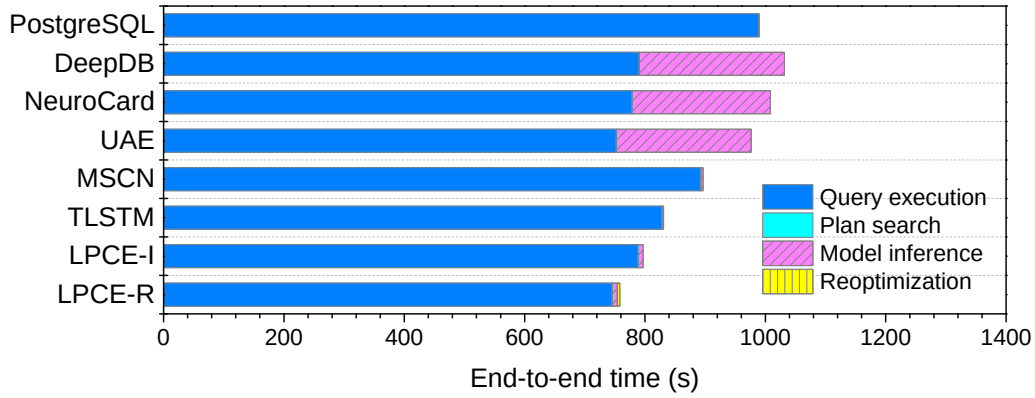
mation. 12 of 22 queries have less than 2 joins, and only 6 queries have over than 4 joins. LPCE-I still contributes to the speedup for query 5, query 8 and query 9 by 1.2x, 1.2x and 1.4x respectively. Though the re-optimization of LPCE-R is triggered for query 8, the refined cardinality does not help further shorten the query. To understand the cardinality estimation on *TPC-ben*, we report estimation q-error of PostgreSQL and compared to *Join-eight* in Table 5.3. At the 90th percentile, the q-error on *TPC-ben* is 12.9, smaller by four orders of magnitude on *Join-eight*. The cardinality estimation on *TPC-ben* is not a hard task for PostgreSQL with its native estimator.

Figure 5.18(b) presents LPCE makes the execution 1.3x faster than PostgreSQL on *TPC-join-six*. Each query of *TPC-join-six* has 6 joins, and the estimation is more challenging than *TPC-ben*. LPCE-I outperforms PostgreSQL because of accurate estimation, and the most benefits come from the adjusted join ordering. LPCE-R further re-optimizes 10 queries, reducing the aggregate end-to-end time execution from 795 to 756s. The data-driven baselines NeuroCard, UAE, and DeepDB perform even worse than PostgreSQL due to the long model inference time. This phenomenon shows that for workloads that cardinality estimation is easy task, short model inference time is key to performance. Compared to query-driven baselines MSCN and TLSTM, LPCE still has shorter end-to-end execution. This is because when the model inference is fast, the estimation accuracy can still have significant impact on end-to-end execution time.

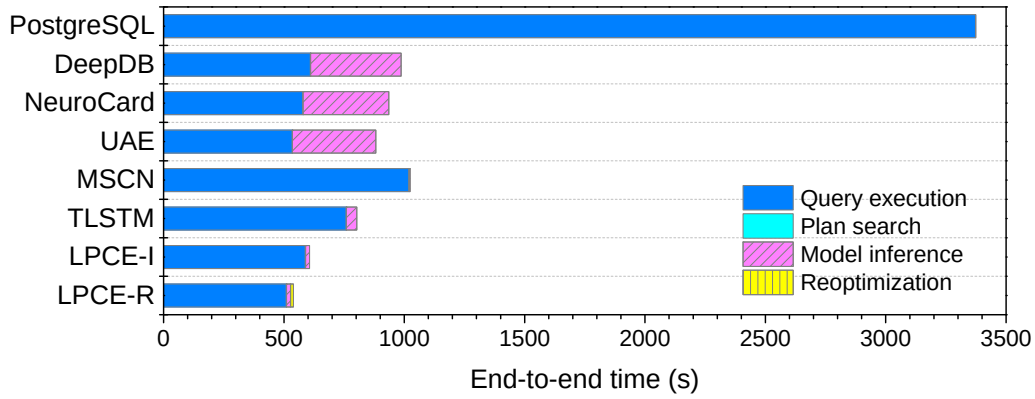
LPCE has significantly shorter aggregate end-to-end time than PostgreSQL on



(a) End-to-end execution time for *TPC-ben* queries.



(b) End-to-end execution time for *TPC-join-six* queries.



(c) End-to-end execution time for *Synthetic-join-eight* queries.

Figure 5.18: End-to-end execution time on varying datasets and workloads.

Synthetic-join-eight, as shown in Figure 5.18(c). Though *Synthetic* is human-generated dataset, its cardinality estimation is challenging due to the high distribution skewness and correlation among columns [106, 202]. The native estimator of PostgreSQL has poor accuracy, and the high accuracy of LPCE-I produces better execution plan. *Synthetic-join-eight* also enjoys the gain of re-optimization with LPCE-R, 27 of 300 queries trigger re-optimization, reducing the end-to-end time from 605 to 538s. Thanks to their good estimation accuracy, the data-driven and hybrid estimators DeepDB, NeuroCard and UAE generally spend short time for query execution. However, their long inference time leads to longer end-to-end time than LPCE. The query-driven baselines MSCN and TLSTM have low the model inference cost, but long query execution time due to worse accuracy. The general performance follows the trend of IMDB, and we conjecture the reason is partly due to the data distribution of *Synthetic* closing to a real world dataset, and partly the more joins (i.e., 8) of queries make estimation significant for quality of execution plan.

5.5.3 Design Choices of LPCE

In this part, we evaluate the designs of LPCE-I and LPCE-R. Recall that LPCE-I has three differences from existing query-driven estimators, i.e., the SRU model, model compression via knowledge distillation, and the node-wise loss function, which we test at first.

SRU model and knowledge distillation: The SRU model and knowledge distillation contribute to the short inference time of LPCE-I because SRU lighter than LSTM, and knowledge distillation compresses LPCE-I to smaller size. We check how the two designs affect inference time and estimation accuracy in Figures 5.19 and 5.20, respectively. LPCE-T adopts the LSTM model while LPCE-S uses the SRU model, and both LPCE-T and LPCE-S are not compressed. LPCE-C directly trains a model with the same size as LPCE-I (i.e., without knowledge distillation) while LPCE-I uses both

Table 5.4: Effect of SRU and knowledge distillation on model size.

Method	LPCE-T	LPCE-S	LPCE-C	LPCE-I
Model size (MB)	37.5	17.9	1.5	1.5

SRU and knowledge distillation. The results (i.e., LPCE-T vs. LPCE-S) show that changing the model from LSTM to simpler SRU has almost no influence on accuracy but speeds up inference by 1.5x on GPU. This is because the SRU model uses fewer matrices and thus reduces the model size to 17.9 MB from 37.5 MB for LSTM, as shown in Table 5.4. With a model of only 1.5 MB, LPCE-C and LPCE-I further speed up LPCE-S by 1.2x on GPU because they use a smaller model. However, LPCE-C has significantly larger estimation error than LPCE-S while LPCE-I matches the accuracy of the full LPCE-S model. This is because the LPCE-C model is over 10x smaller than LPCE-S and thus has weaker ability to learn. However, knowledge distillation effectively helps LPCE-I to learn by fitting the output of the full LPCE-S model.

Figure 5.19 also shows the inference time comparison when using GPU and CPU. GPU can speed up the inference by up to 2.0x compared to CPU. LPCE-T shows the significant improvement with GPU since the LSTM model triggering intensive matrix multiplications can benefit more from the parallelism of GPU. The light weight design of LPCE achieves the speedup on both GPU and CPU. Particularly, the knowledge distillation shows higher efficiency on CPU. This is because CPU has weaker parallelism than GPU, and the decrease of model size can release the pressure of parallelism consumption. However, GPU is still the effective processor for LPCE due to the faster inference time.

Node-wise loss function: Recall that our LPCE-I and LPCE-R are trained with the node-wise loss function while both MSCN and LSTM are trained using the query wise-loss function. We check how the node-wise loss function affects estimation accuracy in Figure 5.21, in which LPCE-Q shares the structure of LPCE-I but adopts

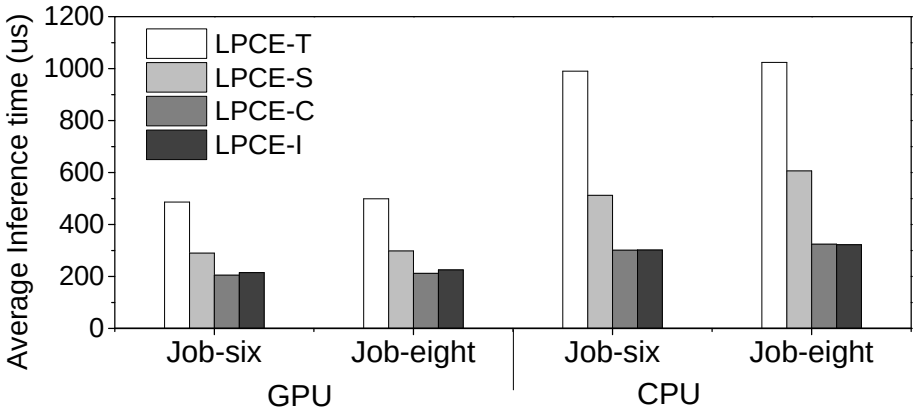
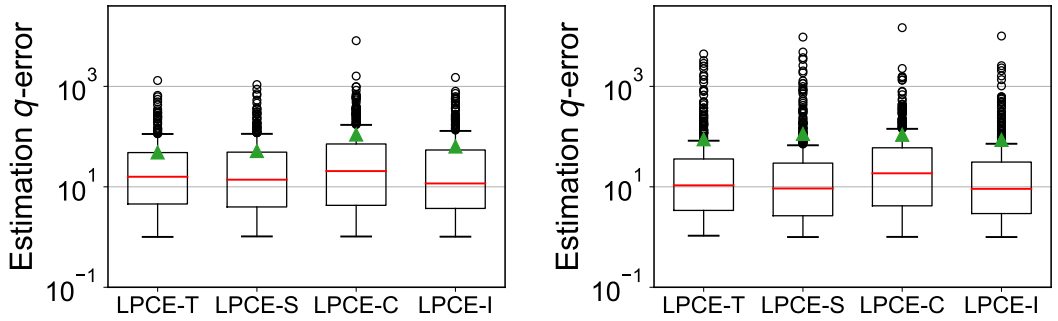


Figure 5.19: Average model inference time for one cardinality on GPU and CPU.

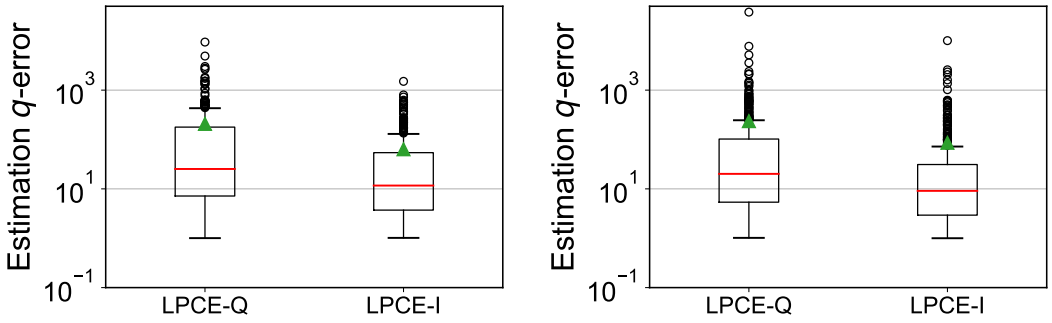


(a) *Join-six* (b) *Join-eight*

Figure 5.20: Effect of SRU and knowledge distillation on estimation accuracy.

the query-wise loss function in Equation 5.2. The results show that using the node-wise loss function significantly improves accuracy, which may be explained from two aspects. First, the node-wise loss function enlarges the training set by considering the estimation errors of all sub-plans and can be regarded as a form of data augmentation. Second, the node-wise loss function allows supervision for all internal nodes in an execution plan, which leads to more informative intermediate representations and consequently better final accuracy.

Design of LPCE-R**:** Recall that our progressive cardinality refinement model LPCE-R uses a hybrid design with three modules. Module cardinality and content are used to embed the executed operators, and their difference is that the cardinality module has



(a) *Join-six* (b) *Join-eight*

Figure 5.21: Effect of node wise loss function on estimation accuracy.

access to the cardinalities of (the result of) the executed operators while the content module does not. The Refine module merges the embedding produced by cardinality and content, and processes the remaining operators (which are not executed) for the final result. We compare LPCE-R with two alternative (perhaps more natural) designs: (1) LPCE-R-*Single* uses only one module sharing the same structure as the cardinality module in LPCE-R, which has access to the cardinalities of intermediate results. The real cardinalities are used for training while for inference the executed operators use the real cardinalities and the remaining operators use the estimated cardinalities. (2) LPCE-R-*Two* uses module cardinality and refine of LPCE-R, in which module cardinality processes the executed operators with real cardinalities while module refine (which does not access cardinalities) takes inputs from module cardinality and processes the remaining operators.

We compare the estimation error of LPCE-R, LPCE-R-*Single* and LPCE-R-*Two* in Table 5.5. The results show that LPCE-R consistently outperforms LPCE-R-*Single* and LPCE-R-*Two*. Among the three models, LPCE-R-*Single* has the worst performance in most cases. This is because LPCE-R-*Single* is trained using the real cardinalities but uses the estimated cardinalities of the remaining operators for inference. As the estimations are inaccurate, errors will accumulate along the join

Table 5.5: q-error of the cardinality estimation provided by different designs of the progressive model for *Join-eight*.

Methods	LPCE-R			LPCE-R-*Single*			LPCE-R-*Two*		
Executed operators	4	8	12	4	8	12	4	8	12
50th	**7.15**	**6.93**	**5.41**	17.1	12.6	8.69	10.0	9.11	7.16
75th	**21.6**	**18.9**	**16.2**	83.1	68.9	45.1	37.0	33.9	25.3
95th	**66.3**	**65.8**	**56.9**	396	314	148	123	102	97.8
99th	**1203**	**1877**	**1561**	5377	5294	2688	3006	2956	1921
max	**8161**	**5743**	**3820**	81022	20536	13139	11778	11223	8133
mean	**72.6**	**67.8**	**51.7**	433	247	156	123	88.3	64.8

tree in the model. LPCE-R-*Two* performs better because its refine module does not rely on cardinality estimations for the remaining operators. Compared with LPCE-R, LPCE-R-*Two* does not use module content, which embeds the contents of the executed operators (e.g., $R.a < 10$ and $R.b > 100$) without using cardinalities. We conjecture that module cardinality tends to focus on the real cardinalities when embedding the executed operators but the contents of the executed operators (such as a selection on one column) are also important as they may influence the remaining operators. Therefore, in LPCE-R, module cardinality complements module content by providing more information about the executed operators.

Training cost: LPCE-I is trained using 10,000 sample queries, and error stabilizes after about 50 epochs for the test queries. Module content and cardinality of LPCE-R are pre-trained using the same procedure as LPCE-I. Module refine is fine-tuned based on module content and cardinality, and can converge within 10 epochs. The total training time of LPCE is about 50 minutes.

We study the training efficiency of LPCE with GPU and CPU in Figure 5.22. The training process conducts model inference on each sample recursively and terminates when the loss value keeps steady. A set of 50 query samples is packaged into a batch,

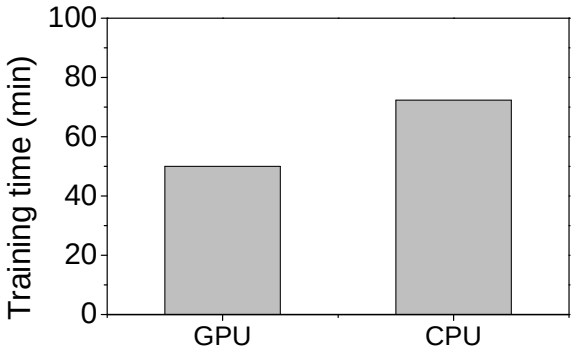


Figure 5.22: Training time with GPU and CPU.

and concurrently processed to make full use of parallelism. GPU provides higher parallelism than CPU to accelerate the training. Hence, we conclude that GPU is an effective processor for LPCE due to both the shorter model inference and training time.

5.6 Related work

In this part, we review works that are most relevant to ours.

Traditional cardinality estimators: *Histogram-based* cardinality estimation methods [45, 5, 73] have been widely used in many industrial database systems as they are simple and have very low overhead. However, they lack the ability to capture the data correlations among the tables as they make the attribute-value-independence assumption. *Sampling-based* approaches [34, 62, 188] outperform histogram-based methods as the correlations in data are naturally captured by data samples. However, sampling-based approaches have two limitations: (i) empty sampling set of join result; and (ii) high sampling overhead. Recent works [215, 123] have proposed index-based and materialized sampling strategies to alleviate the problem of empty result. However, it is still difficult to use sampling to evaluate the cost of all execution

plans as it incurs high space- and time- costs.

Learning-based cardinality estimators: With the advance of machine and deep learning techniques, researchers proposed many learning-based solutions for various problems [113, 114, 134, 112] in the database community. Recently, the database community recognized the potential of replacing traditional cardinality estimation methods by learning-based models (e.g., neural network, autoregressive model) [195, 118]. Existing learning-based estimators can be classified into three categories: *query-driven*, *data-driven* and *hybrid*.

Data-driven cardinality estimators [85, 212] share the same idea with traditional cardinality estimation methods, i.e., they hope to capture the correlations and distributions of data across the tables. DeepDB [85] adopts relational sum product networks (RSPN) to capture the probability distribution among relations, and translates a query into the evaluations of probabilities and expectations based on RSPN. Naru [213] adopts autoregressive models, i.e., masked autoencoder [63] and transformer [187] to estimate the selectivity of equal and range predicates. NeuroCard [212] trains a single deep autoregressive model based on samples collected from the full outer join result of all relations. The model can be used to answer complex queries with joins on any subset of the relations.

Query-driven cardinality estimators [172, 109, 175, 82, 53] formulate cardinality estimation as a regression problem. The contents of queries and their true cardinalities are used as training data to learn a mapping from queries to cardinalities. [52] uses regression techniques such as XGBoost, to train the model to produce approximate cardinality labels. MSCN transforms a query into a feature vector and uses a multi-set convolution network [216] to map it to cardinality estimation. TLSTM processes a query execution plan recursively following its tree structure using LSTM (a kind of RNN) [86].

Hybrid cardinality estimator UAE [200] learns the joint data distribution among the tables as in the data-driven estimators, and uses query samples as auxiliary information at the same time. With an unified deep autoregressive model, UAE learns from data in an unsupervised manner and query samples in a supervised manner.

Existing cardinality estimators cannot achieve *high accuracy* and *fast inference* simultaneously, which are necessary for short end-to-end query execution time. Our LPCE is general framework, which enjoys the fast inference of query-driven estimators and progressively refines estimations to correct large errors. We have shown that data-driven and hybrid estimators also have larger errors for complex queries, and thus may also benefit from our progressive cardinality estimation methodology.

Query re-optimization: Although extensive efforts have been made to improve the accuracy of cardinality estimation, the errors can still be large for complex queries, e.g., those with multiple joins. Query re-optimization techniques [13, 99, 4, 51, 201, 146] have been proposed to combat the influence of large estimation errors on query optimization. The general idea is that the query optimizers first choose an initial execution plan, and then exploit extra knowledge to adjust to a better execution plan. There are two classes of query re-optimization techniques. (1) *Re-optimizing during query execution.* For example, ROX [4] samples data from the intermediate results, and tries different operator implementations and join orders on the samples to adjust the remaining execution plan. (2) *Re-optimizing before query execution.* For example, Wu et al. [201] use a sampling-based method to detect estimation errors and then adjust the execution plan before running it. Our LPCE differs from these query re-optimization techniques in that they are sampling-based while LPCE uses machine learning to exploit information in the executed sub-queries for estimation refinement.

5.7 Chapter Summary

In this section, we conclude the work presented in this chapter, and discuss the related future research directions.

5.7.1 Conclusion

We propose LPCE, a learning-based cardinality estimator that progressively refines the cardinality estimations during the query execution process. GPU is used to support the fast estimation of LPCE. We observed that to reduce the end-to-end execution time of queries, both fast inference and high accuracy are necessary but no existing cardinality estimators can satisfy the two requirements simultaneously. Thus, LPCE adopts a novel framework, which uses a query-driven estimator for fast initial estimation and exploits the executed sub-plans to correct large estimation errors for the remaining operators. We propose techniques including node-wise loss function, knowledge distillation-based model compression and refinement model with three modules to instantiate LPCE. Extensive experiments show that our LPCE outperforms existing learning-based estimators in reducing the end-to-end execution time of queries, especially in worst cases.

5.7.2 Research Directions

There are many interesting problems in applying learning-based progressive cardinality estimation to speed up query execution, including devising tailored plan enumeration algorithms and improving the performance of data-driven or hybrid learning-based estimators. Specifically, we consider future research directions as follows.

First, the policy to judge the benefit of query re-optimization is an interesting and complex issue. In LPCE, we simply set a fixed parameter (i.e., the q-error between

estimated and real cardinalities) to determine re-optimization triggering. The side effect is the case that even when the estimated cardinality is at large error, the execution plan remains unchanged after re-optimization, and the cost of re-optimization instead prolongs the execution. LPCE expects an effective and reliable policy to trigger the re-optimization only when a better execution plan can be guaranteed.

Second, the query re-optimization can be extended to data-driven and hybrid learning-based estimators. The challenges might be what information to extract from the completed subplans, and how to learn the information in a short time.

Third, GPU can be used to speed up the other components of end-to-end execution. In LPCE, GPU is only used to provide fast cardinality estimation. However, the end-to-end execution includes the other components such as plan enumeration. For a complex query involving many tables, the plan space can be huge, and plan search is time-consuming [118]. How to utilize the parallelism of GPU to accelerate the search among a large number of plans is a promising research problem.

Last, LPCE as a deep learning backbone design cannot provide guarantee on the prediction accuracy. LPCE may produce inaccurate estimations for a small fraction of queries when it does not learn the queries from training samples. It is promising to study the prediction uncertainty measurements for the learning-based estimators, and further provide the performance guarantee to the query execution time. The work in this trend would make learning for database more practical in real applications.

Chapter 6

Conclusions and Future Works

In this chapter, we conclude the works presented in this thesis, and discuss the related future works.

6.1 Conclusions

The fast-expanding variety, velocity, and volume of data lead to various challenges on the efficiency of querying and mining data. The new advances of modern hardware open a new horizon to solve the challenges. In this thesis, we propose exploiting modern hardware like GPUs and NVM to optimize three challenging problems on querying and mining data.

The first one is for similarity-based data mining algorithm in Chapter 3. The motivation is to use NVM PIM to tackle the "memory wall" challenge. We identify the bottleneck functions of a given algorithm and offload the most computation to NVM PIM without compromising the accuracy of results. Our work effectively reduces the data transfer overhead, achieving up to 11.0x speedup for state-of-art kNN classification and k-means clustering algorithms.

The second one is for blockchain mining in Chapter 4. The motivation is to utilize the parallelism of NVM PIM for repetitive computation. The mining process requires massive cryptographic hash computation to ensure that the data is secure

and immutable. We map the hash computation onto NVM PIM, and propose the optimization techniques to make full use of the parallelism of NVM PIM. Our work speeds up blockchain mining by up to 778x than CPU-based implementation and up to 3.8x than GPU-based implementation

The third one is for analytical query execution in Chapter 5. The motivation is using GPU to support the fast inference for a deep learning model. We propose LPCE, a learning-based cardinality estimator that progressively refines the cardinality estimations during the query execution process. The information of executed operations is used to refine the estimations for unexecuted ones. We carefully design the estimator to meet both accuracy and efficiency simultaneously. LPCE outperforms existing learning-based estimators and reduces up to 99.7% of end-to-end query execution time.

6.2 Future Works

There are many potential research challenges of speeding up mining and querying on data. We present several opportunities as future research.

Firstly, our work in Chapter 3 only focuses on accelerating the data mining tasks through optimizing the computation of similarity functions on high-dimensional data. We propose three research future directions. Firstly, our method may fail to apply for the tasks that similarity computation on vector data is not involved, such as decision tree-based classification and graph clustering. To speed up these tasks with NVM PIM, new challenges arise such as how to identify the bottleneck, and how to transform the computation to the specific operations supported by NVM PIM. Secondly, the modern hardware such GPU and FPGA has been intensively studied and adopted in real industrial database. For example, PolarDB X-engine at Alibaba utilizes FPGA to accelerate the log-structured tree compaction [87, 27]. While NVM

PIM, GPU and FPGA are not general-purpose processors and superior at various operations, it is interesting to investigate that how to cooperate the each other to compose a powerful heterogeneous computing architecture. Figure 6.1 illustrates a possible design for data mining applications with hierarchical storage. Lastly, our work focuses on utilizing the computing feature of PIM, yet the related techniques such as data compression, data locality during PIM [104, 197] are also very promising for some scenarios. For example, when applying NVM PIM to the session-based recommendation in e-commerce and social media [103], we might organize the data location based on the common dependent processing operations or data features so as to share some computation.

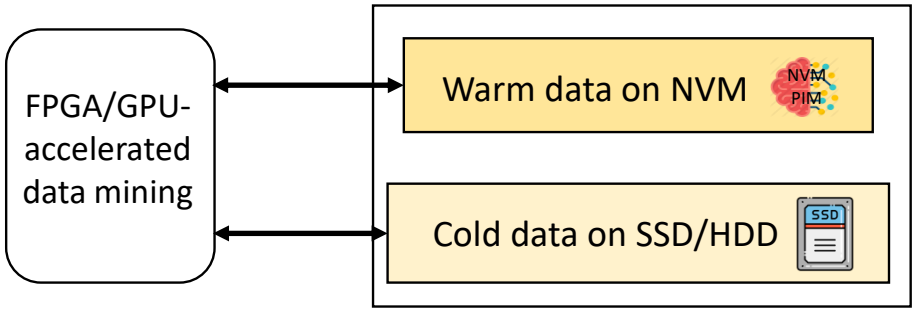


Figure 6.1: An example of cooperating NVM PIM with GPU and FPGA.

Secondly, we develop a solution to speed up the blockchain mining process in Chapter 4. This work focuses on making full use of the high parallelism of NVM PIM. However, NVM PIM has been proven the high potential to reduce the energy consumption for compute-intensive tasks. Blockchain applications typically cost huge energy consumption. Firstly, our next goal is to extend the proposed techniques to achieve an ultra-energy-efficient mining process, and at the same time, the mining speedup is not compromised. Secondly, Blockchain has been widely used in cloud edge computing and Internet-of-things. Such scenarios often include hundreds even thousands devices within in a network. The devices might have different hardware processors such as FPGA and ASIC, providing various computational power. It is

interesting to study how cooperate NVM PIM with the hardware processors, and achieve the efficient proof-of-work in network as shown in Figure 6.2. Lastly, though most applications of using Blockchain now are target on efficient and safe data storage [16], the next step are often mining and analyzing on the data [74]. For example, a bank first collects and stores the transaction records on Blockchain-based network, and then analyzes financial and business reports based on the records. Hence, it is interesting to design a NVM PIM framework that combine both data storage and analysis functionality.

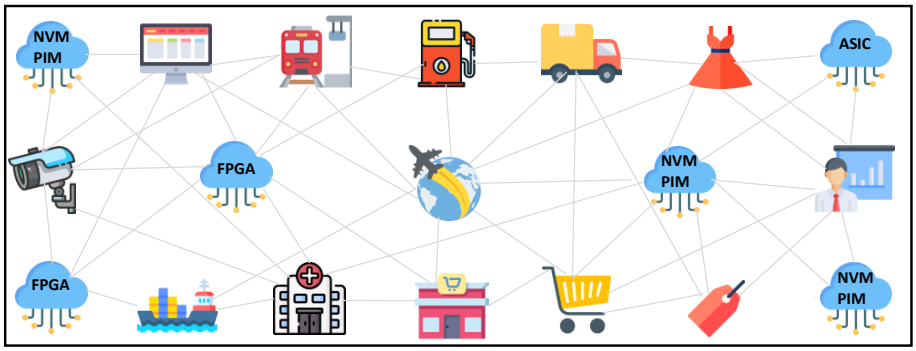


Figure 6.2: An example of using NVM PIM and other hardware in a Blockchain-based network.

Lastly, we propose a progressive learning-based cardinality estimator to optimize the query end-to-end execution in Chapter 5. We consider the three issues as future works. Firstly, an efficient and reliable policy to judge if the query re-optimization is worth triggered. Such policy expects scientific guarantee and works for unique query. The re-optimization is triggered only when we have certainty that the overhead pays off. Secondly, we has shown that learning estimators have various accuracy and estimation overhead. It might be not wisdom that blindly trusts one estimator and uses it all the time. For example, the estimator MSCN might be better choice than NeuroCard for one query, since the it has learned from very similar samples and thus more accurate. We may adopt the multiple estimators as candidates and learn

to identify which one is best choice for given query. Lastly, the recent learning estimators are enhanced along with complicated model design and intricate setting, which requires the computational resource and hardware configuration to provide training and estimation. We think one trend is that commerce cloud offers the user-friendly management as 'model-as-service'. As shown in Figure 6.3, the cardinality estimators as a block box running at cloud devices with sufficient hardware resources, and feed the needs during plan search for query execution at user end.

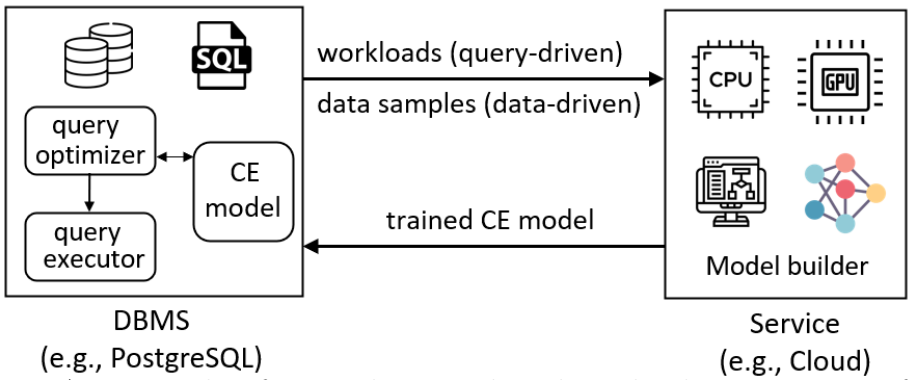


Figure 6.3: An example of using learning-based cardinality estimators for cloud service.

We believe the rapid development of modern hardware always brings new challenges and opportunities on accelerating data mining and querying. The research of exploiting modern hardware is always exciting and meaningful.

Bibliography

[1] PostgreSQL 13.0. https://www.postgresql.org/about/news/postgresql-13-released-2077/.

[2] PAPI 5.7. http://icl.utk.edu/papi/software/.

[3] Nvidia A100. https://https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet.pdf.

[4] Riham Abdel Kader, Peter Boncz, Stefan Manegold, and Maurice Van Keulen. Rox: run-time optimization of xqueries. In *SIGMOD*, pages 615–626, 2009.

[5] Ashraf Aboulnaga and Surajit Chaudhuri. Self-tuning histograms: Building histograms without looking at data. In *SIGMOD*, pages 181–192, 1999.

[6] Cornelius C Agbo, Qusay H Mahmoud, and J Mikael Eklund. Blockchain technology in healthcare: a systematic review. In *Healthcare*, volume 7, page 56, 2019.

[7] Anastassia Ailamaki, David J DeWitt, Mark D Hill, and David A Wood. Dbmss on a modern processor: Where does time go? In *VLDB*, pages 266–277, 1999.

[8] Jasmin Ajanovic. Pci express 3.0 overview. In *Hot Chips Symposium*, pages 1–61, 2009.

[9] Hiroyuki Akinaga and Hisashi Shima. Resistive random access memory (reram) based on metal oxides. *Proceedings of the IEEE*, 98(12):2237–2251, 2010.

[10] Fabien Alibart, Ligang Gao, Brian D Hoskins, and Dmitri B Strukov. High precision tuning of state for memristive devices by adaptable variation-tolerant algorithm. *Nanotechnology*, 23(7):07–52, 2012.

[11] Joshua A Anderson, Chris D Lorenz, and Alex Travesset. General purpose molecular dynamics simulations fully implemented on graphics processing units. *Journal of computational physics*, 227(10):5342–5359, 2008.

[12] Musbah Abdulkarim Musbah Ataya and Musab AM Ali. Acceptance of website security on e-banking. a-review. In *ICSGRC*, pages 201–206, 2019.

[13] Ron Avnur and Joseph M Hellerstein. Eddies: Continuously adaptive query processing. In *SIGMOD*, pages 261–272, 2000.

[14] IG Baek, MS Lee, S Seo, MJ Lee, et al. Highly scalable nonvolatile resistive memory using simple binary oxide driven by asymmetric unipolar voltage pulses. In *IEDM Technical Digest*, pages 587–590, 2004.

[15] Gavina Baralla, Simona Ibba, Michele Marchesi, Roberto Tonelli, and Sebastiano Missineo. A blockchain based system to ensure transparency and reliability in food supply chain. In *European Conference on Parallel Processing*, pages 379–391, 2018.

[16] Nazanin Zahed Benisi, Mehdi Aminian, and Bahman Javadi. Blockchain-based decentralized storage networks: A survey. *JNCA*, 162:102–114, 2020.

[17] Konstantin Bick, Duy Thanh Nguyen, Hyuk-Jae Lee, and Hyun Kim. Fast and accurate memory simulation by integrating dramsim2 into mcsima+. *Electronics*, 7(8):152–162, 2018.

[18] BlazingSQL. `https://blazingsql.com/`.

[19] Johannes Blömer, Christiane Lammersen, Melanie Schmidt, and Christian Sohler. Theoretical analysis of the k-means algorithm–a survey. In *Algorithm Engineering*, pages 81–116. 2016.

[20] Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua A Kroll, and Edward W Felten. Sok: research perspectives and challenges for bitcoin and cryptocurrencies. In *S&P*, pages 104–121, 2015.

[21] Jalil Boukhobza, Stéphane Rubini, Renhai Chen, and Zili Shao. Emerging nvm: A survey on architectural integration and research challenges. *TODAES*, 23(2):1–32, 2017.

[22] Nicolas Bruno, Vivek Narasayya, and Ravi Ramamurthy. Slicing long-running queries. *PVLDB*, 3(1-2):530–541, 2010.

[23] Geoffrey W Burr, Pritish Narayanan, Robert M Shelby, Severin Sidler, Irem Boybat, Carmelo Di Nolfo, and Yusuf Leblebici. Large-scale neural networks implemented with non-volatile memory as the synaptic weight element: Comparative performance analysis (accuracy, speed, and power). In *IEDM*, pages 4–4, 2015.

[24] Geoffrey W Burr, Robert M Shelby, Severin Sidler, Carmelo Di Nolfo, Junwoo Jang, Irem Boybat, Rohit S Shenoy, Pritish Narayanan, Kumar Virwani, Emanuele U Giacometti, et al. Experimental demonstration and tolerancing

of a large-scale neural network (165 000 synapses) using phase-change memory as the synaptic weight element. *IEEE Transactions on Electron Devices*, 62(11):3498–3507, 2015.

[25] CACTI. `http://www.hpl.hp.com/research/cacti/`.

[26] Damla Senol Cali, Gurpreet S Kalsi, Zülal Bingöl, Can Firtina, Lavanya Subramanian, Jeremie S Kim, Rachata Ausavarungnirun, Mohammed Alser, Juan Gomez-Luna, Amirali Boroumand, et al. Genasm: A high-performance, low-power approximate string matching acceleration framework for genome sequence analysis. In *MICRO*, pages 951–966, 2020.

[27] Wei Cao, Yang Liu, Zhushi Cheng, Ning Zheng, Wei Li, Wenjie Wu, Linqiang Ouyang, Peng Wang, Yijing Wang, Ray Kuan, et al. {POLARDB} meets computational storage: Efficiently support analytical workloads in {Cloud-Native} relational database. In *FAST*, pages 29–41, 2020.

[28] Roberto Carboni and Daniele Ielmini. Stochastic memory devices for security and computing. *Advanced Electronic Materials*, 5(9):190–198, 2019.

[29] Kaushik Chakrabarti and Sharad Mehrotra. The hybrid tree: An index structure for high dimensional feature spaces. In *ICDE*, pages 440–447, 1999.

[30] Moses Charikar. Similarity estimation techniques from rounding algorithms. In *STOC*, pages 380–388, 2002.

[31] Ricardo Chaves, Leonel Sousa, Nicolas Sklavos, Apostolos P Fournaris, Georgina Kalogeridou, Paris Kitsos, and Farhana Sheikh. Secure hashing: Sha-1, sha-2, and sha-3. *Circuits and Systems for Security and Privacy*, pages 105–132, 2016.

[32] Pai-Yu Chen, Xiaochen Peng, and Shimeng Yu. Neurosim: A circuit-level macro model for benchmarking neuro-inspired architectures in online learning. *TCAD*, 37(12):3067–3080, 2018.

[33] Yihua Chen, Eric K. Garcia, Maya R. Gupta, Ali Rahimi, and Luca Cazzanti. Similarity-based classification: Concepts and algorithms. *JMLR*, 10:747–776, 2009.

[34] Yu Chen and Ke Yi. Two-level sampling for join size estimation. In *SIGMOD*, pages 759–774, 2017.

[35] Ping Chi, Shuangchen Li, Yuanqing Cheng, Yu Lu, Seung H Kang, and Yuan Xie. Architecture design with stt-ram: Opportunities and challenges. In *ASP-DAC*, pages 109–114, 2016.

[36] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. *ACM SIGARCH Computer Architecture News*, 44(3):27–39, 2016.

[37] Tat-Seng Chua, Jinhui Tang, Richang Hong, Haojie Li, Zhiping Luo, and Yantao Zheng. Nus-wide: a real-world web image database from national university of singapore. In *CIVR*, pages 1–9, 2009.

[38] Michael Crosby, Pradan Pattanayak, Sanjeev Verma, Vignesh Kalyanaraman, et al. Blockchain technology: Beyond bitcoin. *Applied Innovation*, 2(6-10):71, 2016.

[39] Guohao Dai, Tianhao Huang, Yu Wang, Huazhong Yang, and John Wawrzynek. Graphsar: a sparsity-aware processing-in-memory architecture for large-scale graph processing on rerams. In *ASP-DAC*, pages 120–126, 2019.

[40] Dipankar Dasgupta, John M Shrein, and Kishor Datta Gupta. A survey of blockchain from security perspective. *Journal of Banking and Financial Technology*, 3(1):1–17, 2019.

[41] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *SCG*, pages 253–262, 2004.

[42] Rebecca L Davidson and Christopher P Bridges. Error resilient gpu accelerated image processing for space applications. *TPDS*, 29(9):1990–2003, 2018.

[43] Natarajan Deepa, Quoc-Viet Pham, Dinh C Nguyen, Sweta Bhattacharya, B Prabadevi, Thippa Reddy Gadekallu, Praveen Kumar Reddy Maddikunta, Fang Fang, and Pubudu N Pathirana. A survey on blockchain for big data: approaches, opportunities, and future directions. *Future Generation Computer Systems*, 131:209–226, 2022.

[44] Richard Dennis and Gareth Owen. Rep on the block: A next generation reputation system based on the blockchain. In *ICITST*, pages 131–138, 2015.

[45] Amol Deshpande, Minos Garofalakis, and Rajeev Rastogi. Independence is good: Dependency-based histogram synopses for high-dimensional data. *ACM SIGMOD Record*, 30(2):199–210, 2001.

[46] Yufei Ding, Yue Zhao, Xipeng Shen, Madanlal Musuvathi, and Todd Mytkowicz. Yinyang k-means: A drop-in replacement of the classic k-means with consistent speedup. In *ICML*, pages 579–587, 2015.

[47] Tien Tuan Anh Dinh, Ji Wang, Gang Chen, Rui Liu, Beng Chin Ooi, and Kian-Lee Tan. Blockbench: A framework for analyzing private blockchains. In *SIGMOD*, pages 1085–1100, 2017.

[48] Xiangyu Dong, Cong Xu, Yuan Xie, and Norman P Jouppi. Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory. *TCAD*, 31(7):994–1007, 2012.

[49] Jonathan Drake and Greg Hamerly. Accelerated k-means with adaptive distance bounds. In *5th NIPS Workshop on Optimization for Machine Learning*, volume 8, pages 1–4, 2012.

[50] Tuyet Duong, Alexander Chepurnoy, Lei Fan, and Hong-Sheng Zhou. Twinscoin: A cryptocurrency via proof-of-work and proof-of-stake. In *Proceedings of ACM Workshop on Blockchains, Cryptocurrencies, and Contracts*, volume 1, pages 1–13, 2018.

[51] Anshuman Dutt and Jayant R Haritsa. Plan bouquets: query processing without selectivity estimation. In *SIGMOD*, pages 1039–1050, 2014.

[52] Anshuman Dutt, Chi Wang, Vivek Narasayya, and Surajit Chaudhuri. Efficiently approximating selectivity functions using low overhead regression models. *PVLDB*, 13(12):2215–2228, 2020.

[53] Anshuman Dutt, Chi Wang, Azade Nazi, Srikanth Kandula, Vivek Narasayya, and Surajit Chaudhuri. Selectivity estimation for range predicates using lightweight models. *PVLDB*, 12(9):1044–1057, 2019.

[54] Karima Echihabi, Kostas Zoumpatianos, Themis Palpanas, and Houda Benbrahim. The lernaean hydra of data series similarity search: An experimental evaluation of the state of the art. *PVLDB*, 12(2):112–127, 2018.

[55] Muhammad El-Hindi, Carsten Binnig, Arvind Arasu, Donald Kossmann, and Ravi Ramamurthy. Blockchaindb: A shared database on blockchains. *PVLDB*, 12(11):1597–1609, 2019.

[56] Charles Elkan. Using the triangle inequality to accelerate k-means. In *ICML*, pages 147–153, 2003.

[57] Rim Ben Fekih and Mariam Lahami. Application of blockchain technology in healthcare: A comprehensive study. In *International Conference on Smart Homes and Health Telematics*, pages 268–276, 2020.

[58] Md Sadek Ferdous, Mohammad Jabed Morshed Chowdhury, and Mohammad A Hoque. A survey of consensus algorithms in public blockchain systems for crypto-currencies. *JNCA*, 182:1–36, 2021.

[59] Amnah Firdous, Malik M Saad Missen, et al. A highly efficient color image encryption based on linear transformation using chaos theory and sha-2. *MTA*, 78(17):24809–24835, 2019.

[60] Daichi Fujiki, Scott Mahlke, and Reetuparna Das. In-memory data parallel processor. *ACM SIGPLAN Notices*, 53(2):1–14, 2018.

[61] Daniel Fullmer and A Stephen Morse. Analysis of difficulty control in bitcoin and proof-of-work blockchains. In *CDC*, pages 5988–5992, 2018.

[62] Sumit Ganguly, Phillip B. Gibbons, Yossi Matias, and Abraham Silberschatz. Bifocal sampling for skew-resistant join size estimation. In *SIGMOD*, pages 271–281, 1996.

[63] Mathieu Germain, Karol Gregor, Iain Murray, and Hugo Larochelle. MADE: masked autoencoder for distribution estimation. In *ICML*, pages 881–889, 2015.

[64] M Girija, P Manickam, and M Ramaswami. Pripresent: an embedded prime lightweight block cipher for smart devices. *Peer-to-Peer Networking and Applications*, 14(4):2462–2472, 2021.

[65] Maya Gokhale, Bill Holmes, and Ken Iobst. Processing in memory: The terasys massively parallel pim array. *Computer*, 28(4):23–31, 1995.

[66] Vladislav Golyanik, Mitra Nasri, and Didier Stricker. Towards scheduling hard real-time image processing tasks on a single gpu. In *ICIP*, pages 4382–4386, 2017.

[67] Google. An inside look at google bigquery. `https://cloud.google.com/files/BigQueryTechnicalWP.pdf`.

[68] Albert Gordo, Jon Almazán, Jerome Revaud, and Diane Larlus. Deep image retrieval: Learning global representations for image search. In *ECML*, pages 241–257, 2016.

[69] Jianping Gou, Baosheng Yu, Stephen J Maybank, and Dacheng Tao. Knowledge distillation: A survey. *IJCA*, 129(6):1789–1819, 2021.

[70] Patrick JF Groenen and Krzysztof Jajuga. Fuzzy clustering with squared minkowski distances. *Fuzzy Sets and Systems*, 120(2):227–237, 2001.

[71] Robert L Grossman, Chandrika Kamath, Philip Kegelmeyer, Vipin Kumar, and Raju Namburu. *Data mining for scientific and engineering applications*, volume 2. Springer Science & Business Media, 2013.

[72] Ramzi Guesmi, Mohamed Amine Ben Farah, Abdennaceur Kachouri, and Mounir Samet. A novel chaos-based image encryption using dna sequence operation and secure hash algorithm sha-2. *Nonlinear Dynamics*, 83(3):1123–1136, 2016.

[73] Dimitrios Gunopulos, George Kollios, Vassilis J Tsotras, and Carlotta Domeniconi. Selectivity estimators for multidimensional range queries over real attributes. *The VLDB Journal*, 14(2):137–154, 2005.

[74] Ye Guo and Chen Liang. Blockchain application and outlook in the banking industry. *Financial innovation*, 2(1):1–12, 2016.

[75] Anika Gupta and Deepak Garg. Applying data mining techniques in job recommender system for considering candidate job preferences. In *ICACCI*, pages 1458–1465, 2014.

[76] Diksha Gupta, Jared Saia, and Maxwell Young. Proof of work without all the work. In *ICPS*, pages 1–10, 2018.

[77] Saransh Gupta, Mohsen Imani, Behnam Khaleghi, Venkatesh Kumar, and Tajana Rosing. Rapid: A reram processing in-memory architecture for dna sequence alignment. In *ISLPED*, pages 1–6, 2019.

[78] Greg Hamerly and Jonathan Drake. Accelerating lloyds algorithm for k-means clustering. In *Partitional Clustering Algorithms*, pages 41–78. 2015.

[79] Jiawei Han, Hong Cheng, Dong Xin, and Xifeng Yan. Frequent pattern mining: current status and future directions. *DMKD*, 15(1):55–86, 2007.

[80] Jiawei Han, Jian Pei, and Micheline Kamber. *Data mining: concepts and techniques*. Morgan Kaufmann, 2011.

[81] Lei Han, Zhaoyan Shen, Zili Shao, H Howie Huang, and Tao Li. A novel reram-based processing-in-memory architecture for graph computing. In *NVMSA*, pages 1–6, 2017.

[82] Shohedul Hasan, Saravanan Thirumuruganathan, Jees Augustine, Nick Koudas, and Gautam Das. Deep learning models for selectivity estimation of multi-attribute queries. In *SIGMOD*, pages 1035–1050, 2020.

[83] HeavyDB. `https://www.heavy.ai/`.

[84] Max Heimel, Martin Kiefer, and Volker Markl. Self-tuning, gpu-accelerated kernel density models for multidimensional selectivity estimation. In *SIGMOD*, pages 1477–1492, 2015.

[85] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. Deepdb: Learn from data, not from queries! *PVLDB*, 13(7):992–1005, 2020.

[86] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[87] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. X-engine: An optimized storage engine for large-scale e-commerce transaction processing. In *SIGMOD*, pages 651–665, 2019.

[88] Wenqin Huangfu, Shuangchen Li, Xing Hu, and Yuan Xie. Radar: a 3d-reram based dna alignment accelerator architecture. In *DAC*, pages 1–6, 2018.

[89] Seyoung Huh, Sangrae Cho, and Soohyung Kim. Managing iot devices using blockchain platform. In *ICACT*, pages 464–467, 2017.

[90] Yoonho Hwang, Bohyung Han, and Hee-Kap Ahn. A fast nearest neighbor search algorithm by nonlinear embedding. In *CVPR*, pages 3053–3060, 2012.

[91] Mohsen Imani, Saransh Gupta, and Tajana Rosing. Ultra-efficient processing in-memory for data intensive applications. In *DAC*, pages 1–6, 2017.

[92] Mohsen Imani, Saransh Gupta, Sahil Sharma, and Tajana Rosing. Nvquery: Efficient query processing in non-volatile memory. *TCAD*, 38(4):628–639, 2018.

[93] Mohsen Imani, Yeseong Kim, and Tajana Rosing. Mpim: Multi-purpose in-memory processing using configurable resistive memory. In *ASP-DAC*, pages 757–763, 2017.

[94] NVIDIA System Management Interface. `https://developer.nvidia.com/nvidia-system-management-interface`.

[95] Anil K. Jain. Data clustering: 50 years beyond k-means. *Pattern Recognition Letters*, 31(8):651–666, 2010.

[96] Shubham Jain, Ashish Ranjan, Kaushik Roy, and Anand Raghunathan. Computing in memory with spin-transfer torque magnetic ram. *VLSI*, 26(3):470–483, 2017.

[97] Shubham Jain, Sachin Sapatnekar, Jian-Ping Wang, Kaushik Roy, and Anand Raghunathan. Computing-in-memory with spintronics. In *DATE*, pages 1640–1645, 2018.

[98] Herve Jegou, Matthijs Douze, and Cordelia Schmid. Hamming embedding and weak geometric consistency for large scale image search. In *ECML*, pages 304–317, 2008.

[99] Navin Kabra and David J DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *SIGMOD*, pages 106–117, 1998.

[100] Konstantinos Karanasos, Andrey Balmin, Marcel Kutsch, Fatma Ozcan, Vuk Ercegovac, Chunyang Xia, and Jesse Jackson. Dynamically optimizing queries over large scale data platforms. In *SIGMOD*, pages 943–954, 2014.

[101] Tomas Karnagel, René Müller, and Guy M. Lohman. Optimizing gpu-accelerated group-by and aggregation. *ADMS@VLDB*, 8:20, 2015.

[102] Norio Katayama and Shin'ichi Satoh. The sr-tree: An index structure for high-dimensional nearest neighbor queries. *ACM SIGMOD Record*, 26(2):369–380, 1997.

[103] Liu Ke, Udit Gupta, Benjamin Youngjae Cho, David Brooks, Vikas Chandra, Utku Diril, Amin Firoozshahian, Kim Hazelwood, Bill Jia, Hsien-Hsin S Lee, et al. Recnmp: Accelerating personalized recommendation with near-memory processing. In *ISCA*, pages 790–803, 2020.

[104] Ken Kennedy and Kathryn S McKinley. Optimizing for parallelism and data locality. In *ICS*, pages 323–334, 1992.

[105] Tiago Rodrigo Kepe, Eduardo Cunha de Almeida, and Marco A. Z. Alves. Database processing-in-memory: an experimental study. *PVLDB*, 13(3):334–347, 2019.

[106] Kyoungmin Kim, Jisung Jung, In Seo, Wook-Shin Han, Kangwoo Choi, and Jaehyok Chong. Learned cardinality estimation: An in-depth study. In *SIGMOD*, pages 1214–1227, 2022.

[107] Hideaki Kimura. Foedus: Oltp engine for a thousand cores and nvram. In *SIGMOD*, pages 691–706, 2015.

[108] Kinetica. `https://www.kinetica.com/`.

[109] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. Learned cardinalities: Estimating correlated joins with deep learning. In *CIDR*, 2019.

[110] Bryan Klimt and Yiming Yang. The enron corpus: A new dataset for email classification research. In *ECML*, pages 217–226, 2004.

[111] Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *S&P*, pages 839–858, 2016.

[112] Tim Kraska, Mohammad Alizadeh, Alex Beutel, H Chi, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. Sagedb: A learned database system. In *CIDR*, 2019.

[113] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *SIGMOD*, pages 489–504, 2018.

[114] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph Hellerstein, and Ion Stoica. Learning to optimize join queries with d reinforcement learning. *arXiv preprint arXiv:1808.03196*, 2018.

[115] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*, pages 1097–1105, 2012.

[116] Philip J Kuekes, Gregory S Snider, and R Stanley Williams. Crossbar nanocomputers. *Scientific American*, 293(5):72–80, 2005.

[117] Shahar Kvatinsky, Dmitry Belousov, Slavik Liman, Guy Satat, Nimrod Wald, Eby G Friedman, Avinoam Kolodny, and Uri C Weiser. Magicmemristor-aided logic. *TCAS-II*, 61(11):895–899, 2014.

[118] Hai Lan, Zhifeng Bao, and Yuwei Peng. A survey on advancing the dbms query optimizer: Cardinality estimation, cost model, and plan enumeration. *Data Science and Engineering*, pages 1–16, 2021.

[119] Manuel Le Gallo, Abu Sebastian, Giovanni Cherubini, Heiner Giefers, and Evangelos Eleftheriou. Compressed sensing recovery using computational memory. In *IEDM*, pages 28–3, 2017.

[120] Manuel Le Gallo, Abu Sebastian, Roland Mathis, Matteo Manica, Heiner Giefers, Tomas Tuma, Costas Bekas, Alessandro Curioni, and Evangelos Eleftheriou. Mixed-precision in-memory computing. *Nature Electronics*, 1(4):246–253, 2018.

[121] Tao Lei, Yu Zhang, Sida I. Wang, Hui Dai, and Yoav Artzi. Simple recurrent units for highly parallelizable recurrence. In *EMNLP*, pages 4470–4481, 2018.

[122] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *PVLDB*, 9(3):204–215, 2015.

[123] Viktor Leis, Bernhard Radke, Andrey Gubichev, Alfons Kemper, and Thomas Neumann. Cardinality estimation done right: Index-based join sampling. In *CIDR*, 2017.

[124] David Levinthal. Performance analysis guide for intel core i7 processor and intel xeon 5500 processors. *Intel Performance Analysis Guide*, 30:18, 2009.

[125] Yi-Ching Liaw, Maw-Lin Leou, and Chien-Min Wu. Fast exact k nearest neighbors search using an orthogonal search tree. *Pattern Recognition*, 43(6):2351–2358, 2010.

[126] Iuon-Chang Lin and Tzu-Chun Liao. A survey of blockchain security issues and challenges. *International Journal of Network Security*, 19(5):653–659, 2017.

[127] Jie Liu, Wenqian Dong, Dong Li, and Qingqing Zhou. Fauce: Fast and accurate deep ensembles with uncertainty for cardinality estimation. *PVLDB*, 14(11):1950–1963, 2021.

[128] Tz-Yi Liu, Tian Hong Yan, Roy Scheuerlein, and et al. A 130.7mm^2 2-layer 32gb reram memory device in 24nm technology. In *ISSCC*, pages 210–211, 2013.

[129] Gabriel H Loh. 3d-stacked memory architectures for multi-core processors. *ACM SIGARCH Computer Architecture News*, 36(3):453–464, 2008.

[130] Guy Lohman. Is query optimization a "solve" problem. In *Proceeding Workshop on Database Query Optimization*, volume 13, page 10, 2014.

[131] Qinghua Lu and Xiwei Xu. Adaptable blockchain-based systems: A case study for product traceability. *IEEE Software*, 34(6):21–27, 2017.

[132] Yi Lu, Shiyong Lu, Farshad Fotouhi, Youping Deng, and Susan J Brown. Incremental genetic k-means algorithm and its application in gene expression data analysis. *BMC Bioinformatics*, 5(1):1–10, 2004.

[133] Falei Luo, Shanshe Wang, Shiqi Wang, Xinfeng Zhang, Siwei Ma, and Wen Gao. Gpu-based hierarchical motion estimation for high efficiency video coding. *TMM*, 21(4):851–862, 2018.

[134] Ryan C. Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. Neo: A learned query optimizer. *PVLDB*, 12(11):1705–1718, 2019.

[135] Volker Markl, Vijayshankar Raman, David Simmen, Guy Lohman, Hamid Pirahesh, and Miso Cilimdzic. Robust query processing through progressive optimization. In *SIGMOD*, pages 659–670, 2004.

[136] Raffaele Martino and Alessandro Cilardo. Sha-2 acceleration meeting the needs of emerging applications: A comparative survey. *IEEE Access*, 8:28415–28436, 2020.

[137] Brian McFee, Thierry Bertin-Mahieux, Daniel PW Ellis, and Gert RG Lanckriet. The million song dataset challenge. In *WWW*, pages 909–916, 2012.

[138] Marvin Minsky and Seymour Papert. Perceptron: an introduction to computational geometry. *The MIT Press*, 19(88):2, 1969.

[139] Sushmita Mitra and Tinku Acharya. *Data mining: multimedia, soft computing, and bioinformatics*. John Wiley & Sons, 2005.

[140] Sparsh Mittal. A survey of reram-based architectures for processing-in-memory and neural networks. *Machine learning and knowledge extraction*, 1(1):75–114, 2019.

[141] Shin Morishima and Hiroki Matsutani. Accelerating blockchain search of full nodes using gpus. In *PDP*, pages 244–248, 2018.

[142] Abdullah Mueen. Time series motif discovery: dimensions and applications. *DMKD*, 4(2):152–159, 2014.

[143] Marius Muja and David G Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. *VISAPP*, 2(331-340):2, 2009.

[144] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review*, page 21260, 2008.

[145] SR Nandakumar, Manuel Le Gallo, Irem Boybat, Bipin Rajendran, Abu Sebastian, and Evangelos Eleftheriou. Mixed-precision architecture based on computational memory for training deep neural networks. In *ISCAS*, pages 1–5, 2018.

[146] Thomas Neumann and Cesar Galindo-Legaria. Taking the edge off cardinality estimation errors using incremental execution. *Datenbanksysteme für Business, Technologie und Web*, P-214:73–92, 2013.

[147] Dimin Niu, Cong Xu, Naveen Muralimanohar, Norman P Jouppi, and Yuan Xie. Design of cross-point metal-oxide reram emphasizing reliability and cost. In *ICCAD*, pages 17–23, 2013.

[148] Ohm's and Kirchhoffs Laws. `https://78bbm3rv7ks4b6i8j3cuklc1-wpengine.netdna-ssl.com/wp-content/uploads/tutoring/handouts/Ohms-and-Kirchoffs-Laws.pdf`.

[149] Ismail Oukid and Wolfgang Lehner. Data structure engineering for byte-addressable non-volatile memory. In *SIGMOD*, pages 1759–1764, 2017.

[150] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[151] Yu Pan, Peng Ouyang, Yinglin Zhao, Wang Kang, Shouyi Yin, Youguang Zhang, Weisheng Zhao, and Shaojun Wei. A multilevel cell stt-mram-based computing in-memory accelerator for binary convolutional neural network. *IEEE Transactions on Magnetics*, 54(11):1–5, 2018.

[152] Farhana Parveen, Zhezhi He, Shaahin Angizi, and Deliang Fan. Hielm: Highly flexible in-memory computing using stt mram. In *ASP-DAC*, pages 361–366, 2018.

[153] David A Patterson and John L Hennessy. *Computer organization and design ARM edition: the hardware software interface*. Morgan kaufmann, 2016.

[154] Guido Perboli, Stefano Musso, and Mariangela Rosano. Blockchain in logistics and supply chain: A lean approach for designing real-world use cases. *IEEE Access*, 6:62018–62028, 2018.

[155] Gareth W Peters and Efstathios Panayi. Understanding modern banking ledgers through blockchain technologies: Future of transaction processing and smart contracts on the internet of money. In *Banking Beyond Banks and Money*, pages 239–278. 2016.

[156] Matt Poremba, Sparsh Mittal, Dong Li, Jeffrey S Vetter, and Yuan Xie. Destiny: A tool for modeling emerging 3d nvm and edram caches. In *DATE*, pages 1543–1546, 2015.

[157] Presto. Long-running queries have been major pain points for presto. `https://prestodb.io/blog/2019/08/05/presto-unlimited-mpp-database-at-scale`.

[158] Xuedi Qin, Yuyu Luo, Nan Tang, and Guoliang Li. Making data visualization more efficient and effective: a survey. *The VLDB Journal*, 29(1):93–117, 2020.

[159] Nvidia GeForce GTX 1080 Ti Review. `https://bit-tech.net/reviews/tech/graphics/nvidia-geforce-gtx-1080-ti-review/1/`.

[160] Kohei Nakamura Sakakibara, Yuma and Hiroki Matsutani. An fpga nic based hardware caching for blockchain. In *HEART*, pages 11–16, 2017.

[161] Abu Sebastian, Manuel Le Gallo, Geoffrey W Burr, Sangbum Kim, Matthew BrightSky, and Evangelos Eleftheriou. Tutorial: Brain-inspired computing using phase-change memory devices. *Journal of Applied Physics*, 124(11):111101, 2018.

[162] Abu Sebastian, Tomas Tuma, Nikolaos Papandreou, Manuel Le Gallo, Lukas Kull, Thomas Parnell, and Evangelos Eleftheriou. Temporal correlation detection using computational phase-change memory. *Nature Communications*, 8(1):1–10, 2017.

[163] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access path selection in a relational database management system. In *SIGMOD*, pages 23–34, 1979.

[164] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R Stanley Williams, and Vivek Srikumar. Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. *ACM SIGARCH Computer Architecture News*, 44(3):14–26, 2016.

[165] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. A study of the fundamental performance characteristics of gpus and cpus for database analytics. In *SIGMOD*, pages 1617–1632, 2020.

[166] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. Nexus: a gpu cluster engine for accelerating dnn-based video analysis. In *SOSP*, pages 322–337, 2019.

[167] Panagiotis Sioulas, Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. Hardware-conscious hash-joins on gpus. In *ICDE*, pages 698–709, 2019.

[168] Linghao Song, Xuehai Qian, Hai Li, and Yiran Chen. Pipelayer: A pipelined reram-based accelerator for deep learning. In *HPCA*, pages 541–552, 2017.

[169] Linghao Song, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. Graphr: Accelerating graph processing using reram. In *HPCA*, pages 531–543, 2018.

[170] Michael Spain, Sean Foley, and Vincent Gramoli. The impact of ethereum throughput and fees on transaction latency during icos. In *International Conference on Blockchain Economics, Security and Protocols*, pages 9:1–9:15, 2020.

[171] Jacob R Stevens, Ashish Ranjan, Dipankar Das, Bharat Kaul, and Anand Raghunathan. Manna: An accelerator for memory-augmented neural networks. In *MICRO*, pages 794–806, 2019.

[172] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. LEO - db2's learning optimizer. In *VLDB*, pages 19–28, 2001.

[173] John E Stone, James C Phillips, Peter L Freddolino, David J Hardy, Leonardo G Trabuco, and Klaus Schulten. Accelerating molecular modeling applications with graphics processors. *Journal of Computational Chemistry*, 28(16):2618–2640, 2007.

[174] Dmitri B Strukov, Gregory S Snider, Duncan R Stewart, and R Stanley Williams. The missing memristor found. *Nature*, 453(7191):80, 2008.

[175] Ji Sun and Guoliang Li. An end-to-end learning-based cost estimator. *PVLDB*, 13(3):307–319, 2019.

[176] Yuliang Sun, Yu Wang, and Huazhong Yang. Energy-efficient sql query exploiting rram-based process-in-memory structure. In *NVMSA*, pages 1–6, 2017.

[177] Christina Teflioudi, Rainer Gemulla, and Olga Mykytiuk. LEMP: fast retrieval of large entries in a matrix product. In *SIGMOD*, pages 107–122, 2015.

[178] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. Collecting performance data with papi-c. In *Tools for High Performance Computing*, pages 157–173. 2010.

[179] Anita Tino, Caroline Collange, and André Seznec. Simt-x: Extending single-instruction multi-threading to out-of-order cores. *TACO*, 17(2):1–23, 2020.

[180] Diego G Tomé, Tim Gubner, Mark Raasveldt, Eyal Rozenberg, and Peter A Boncz. Optimizing group-by and aggregation using gpu-cpu co-processing. In *ADMS@VLDB*, pages 1–10, 2018.

[181] Antonio Torralba, Robert Fergus, and William T. Freeman. 80 million tiny images: A large data set for nonparametric object and scene recognition. *TPAMI*, 30(11):1958–1970, 2008.

[182] TPC-H. `https://www.tpc.org/tpch/default.asp`.

[183] Cole Trapnell and Michael C Schatz. Optimizing data intensive gpgpu computations for dna sequence alignment. *Parallel Computing*, 35(8-9):429–440, 2009.

[184] UCI. `https://archive.ics.uci.edu/ml/datasets/yearpredictionmsd`.

[185] Intel Xeon Processor E5-2620 v4. `https://www.intel.com/content/www/us/en/products/sku/92986/intel-xeon-processor-e52620-v4-20m-cache-2-10-ghz/specifications.html`.

[186] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. Managing non-volatile memory in database systems. In *SIGMOD*, pages 1541–1555, 2018.

[187] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NIPS*, pages 5998–6008, 2017.

[188] David Vengerov, Andre Cavalheiro Menck, Mohamed Zaït, and Sunil Chakkappen. Join size estimation subject to filter conditions. *PVLDB*, 8(12):1530–1541, 2015.

[189] Haris Volos, Guilherme Magalhaes, Ludmila Cherkasova, and Jun Li. Quartz: A lightweight performance emulator for persistent memory software. In *MERC*, pages 37–49, 2015.

[190] Ari Ezra Waldman. Privacy, sharing, and trust: The facebook study. *Case Western Reserve Law Review*, 67:193, 2016.

[191] Fang Wang, Zhaoyan Shen, Lei Han, and Zili Shao. Reram-based processing-in-memory architecture for blockchain platforms. In *ASP-DAC*, pages 615–620, 2019.

[192] Fang Wang, Xiao Yan, Man Lung Yiu, Shuai Li, Zunyao Mao, and Tang Bo. Speeding up end-to-end query execution via learning-based progressive cardinality estimation. In *Submitted to SIGMOD*.

[193] Fang Wang, Man Lung Yiu, and Zili Shao. Accelerating similarity-based mining tasks on high-dimensional data by processing-in-memory. In *ICDE*, pages 1859–1864, 2021.

[194] Jason TL Wang, Mohammed J Zaki, Hannu TT Toivonen, and Dennis Shasha. Introduction to data mining in bioinformatics. In *Data Mining in Bioinformatics*, pages 3–8. 2005.

[195] Xiaoying Wang, Changbo Qu, Weiyuan Wu, Jiannan Wang, and Qingqing Zhou. Are we ready for learned cardinality estimation? *PVLDB*, 14(9):1640–1654, 2021.

[196] Simon AJ Winder and Matthew Brown. Learning local image descriptors. In *CVPR*, pages 1–8, 2007.

[197] Michael E Wolf and Monica S Lam. A data locality optimizing algorithm. In *PLDI*, pages 30–44, 1991.

[198] H-S Philip Wong, Simone Raoux, SangBum Kim, Jiale Liang, John P Reifenberg, Bipin Rajendran, Mehdi Asheghi, and Kenneth E Goodson. Phase change memory. *Proceedings of the IEEE*, 98(12):2201–2227, 2010.

[199] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 151(2014):1–32, 2014.

[200] Peizhi Wu and Gao Cong. A unified deep model of learning from both data and queries for cardinality estimation. In *SIGMOD*, pages 2009–2022, 2021.

[201] Wentao Wu, Jeffrey F Naughton, and Harneet Singh. Sampling-based query re-optimization. In *SIGMOD*, pages 1721–1736, 2016.

[202] Ziniu Wu, Amir Shaikhha, Rong Zhu, Kai Zeng, Yuxing Han, and Jingren Zhou. Bayescard: Revitilizing bayesian frameworks for cardinality estimation. *arXiv preprint arXiv:2012.14743*, 2020.

[203] Lixue Xia, Boxun Li, Tianqi Tang, Peng Gu, Pai-Yu Chen, Shimeng Yu, Yu Cao, Yu Wang, Yuan Xie, and Huazhong Yang. MNSIM: simulation platform for memristor-based neuromorphic computing system. *TCAD*, 37(5):1009–1022, 2018.

[204] Yao Xiao, Shahin Nazarian, and Paul Bogdan. Prometheus: Processing-in-memory heterogeneous architecture design from a multi-layer network theoretic strategy. In *DATE*, pages 1387–1392, 2018.

[205] Zhe Xiao, Zengxiang Li, Yong Liu, Ling Feng, Weiwen Zhang, Thanarit Lertwuthikarn, and Rick Siow Mong Goh. Emrshare: A cross-organizational medical data sharing and management framework using permissioned blockchain. In *ICPADS*, pages 998–1003, 2018.

[206] Zehui Xiong, Yang Zhang, Dusit Niyato, Ping Wang, and Zhu Han. When mobile blockchain meets edge computing: challenges and applications. *IEEE Communications Magazine*, 56(8):33–39, 2018.

[207] Cong Xu, Dimin Niu, Naveen Muralimanohar, Rajeev Balasubramonian, Tao Zhang, Shimeng Yu, and Yuan Xie. Overcoming the challenges of crossbar resistive memory architectures. In *HPCA*, pages 476–488, 2015.

[208] Cong Xu, Dimin Niu, Naveen Muralimanohar, Norman P Jouppi, and Yuan Xie. Understanding the trade-offs in multi-level cell reram memory design. In *DAC*, pages 1–6, 2013.

[209] Xiwei Xu, Qinghua Lu, Yue Liu, Liming Zhu, Haonan Yao, and Athanasios V Vasilakos. Designing blockchain-based applications a case study for imported product traceability. *Future Generation Computer Systems*, 92:399–406, 2019.

[210] Makoto Yabuta, Anh Nguyen, Shinpei Kato, Masato Edahiro, and Hideyuki Kawashima. Relational joins on gpus: A closer look. *TPDS*, 28(9):2663–2673, 2017.

[211] Zongheng Yang, Wei-Lin Chiang, Sifei Luan, Gautam Mittal, Michael Luo, and Ion Stoica. Balsa: Learning a query optimizer without expert demonstrations. In *SIGMOD*, pages 931–944, 2022.

[212] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. Neurocard: One cardinality estimator for all tables. *PVLDB*, 14(1):61–73, 2020.

[213] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Xi Chen, Pieter Abbeel, Joseph M. Hellerstein, Sanjay Krishnan, and Ion Stoica. Deep unsupervised cardinality estimation. *PVLDB*, 13(3):279–292, 2019.

[214] Byoung-Kee Yi and Christos Faloutsos. Fast time sequence indexing for arbitrary lp norms. In *VLDB*, pages 385–394, 2000.

[215] Feng Yu, Wen-Chi Hou, Cheng Luo, Dunren Che, and Mengxia Zhu. CS2: a new database synopsis for query estimation. In *SIGMOD*, pages 469–480, 2013.

[216] Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Ruslan Salakhutdinov, and Alexander Smola. Deep sets. In *NIPS*, pages 3391–3401, 2017.

[217] Furqan Zahoor, Tun Zainal Azni Zulkifli, and Farooq Ahmad Khanday. Resistive random access memory (rram): an overview of materials, switching mechanism, performance, multilevel cell (mlc) storage, modeling, and applications. *Nanoscale Research Letters*, 15(1):1–26, 2020.

[218] Yu Zhang and Jiangtao Wen. An iot electric business model based on the protocol of bitcoin. In *ICIN*, pages 184–191, 2015.

[219] Xuesong Zhao and Kaifan Ji. Tourism e-commerce recommender system based on web data mining. In *International Conference on Computer Science & Education*, pages 1485–1488, 2013.

[220] Yang Zheng, Cong Xu, and Yuan Xie. Modeling framework for cross-point resistive memory design emphasizing reliability and variability issues. In *ASP-DAC*, pages 112–117, 2015.

[221] Minxuan Zhou, Mohsen Imani, Saransh Gupta, Yeseong Kim, and Tajana Rosing. Gram: graph processing in a reram-based computational memory. In *ASP-DAC*, pages 591–596, 2019.

[222] Pengfei Zuo, Yu Hua, and Jie Wu. Write-optimized and high-performance hashing index scheme for persistent memory. In *OSDI*, pages 461–476, 2018.

[223] Guy Zyskind and Oz Nathan. Decentralizing privacy: Using blockchain to protect personal data. In *SPW*, pages 180–184, 2015.