**Title**

Data-driven Performance Optimization for Data-intensive Applications



**Author**

Liu, Jie

**Publication Date**

2024

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA, MERCED

**Data-driven Performance Optimization for Data-intensive Applications**

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Electrical Engineering and Computer Science

by

Jie Liu

Committee in charge:

Dong Li, Chair
Florin Rusu
Pengfei Su

Spring 2024

The dissertation of Jie Liu is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

---

Professor Dong Li, Chair

---

Professor Florin Rusu

---

Professor Pengfei Su

University of California, Merced

Spring 2024

DEDICATION

To my family.

TABLE OF CONTENTS


Signature Page . . . iii

Dedication . . . iv

Table of Contents . . . v

List of Figures . . . viii

List of Tables . . . xi

Acknowledgements . . . xii

Vita and Publications . . . xiii

Abstract . . . xiii

Chapter 1 Introduction . . . 1
1.1 Primary Contributions . . . 4
1.2 Outline and Previously Published Work . . . 9

Chapter 2 Background . . . 11
2.1 Background on Mobile Processors . . . 11
2.2 Background of DNNs Training . . . 12

Chapter 3 Flame: A Self-Adaptive Auto-Labeling System for Heterogeneous Mobile Processors . . . 15
3.1 Overview . . . 15
3.2 Model Design . . . 16
3.2.1 Labeling Functions Generation . . . 17
3.2.2 Labeling Functions Self-Adaption . . . 20
3.2.3 Labeling Results Guarantees . . . 22
3.3 System Design . . . 23
3.3.1 Leverage GPU . . . 24
3.3.2 Leverage DSP and CPU . . . 26
3.3.3 Implementation . . . 28
3.4 Evaluation . . . 29
3.4.1 Labeling Quality . . . 31
3.4.2 Analysis on Execution Time . . . 33
3.4.3 Analysis on Energy Consumption . . . 34
3.4.4 Micro-Benchmarking Results . . . 36
3.4.5 Evaluation on User Experience . . . 37
3.5 Summary . . . 39

Chapter 4 Fauce: Fast and Accurate Deep Ensembles with Uncertainty for Cardinality Estimation . . . 40
4.1 Overview of Fauce . . . 40
4.2 Problem description . . . 41
4.2.1 Notations . . . 41
4.2.2 Formulation as a Regression Problem . . . 42
4.3 Query Featurization . . . 42
4.3.1 Tables and Joins Encoding . . . 43
4.3.2 Columns Encoding . . . 45
4.3.3 Range Representation . . . 46
4.4 Choice of regression methods . . . 47
4.4.1 Cardinality Transformation . . . 48
4.4.2 Training Data Generation . . . 49
4.5 Model Design . . . 50
4.5.1 Uncertainty Quantification . . . 51
4.5.2 Training and Inference . . . 52
4.5.3 Management of Estimation Uncertainty . . . 53
4.5.4 Integration with DBMS . . . 55
4.6 Evaluation . . . 56
4.6.1 Experimental Setup . . . 56
4.6.2 Estimation Quality . . . 58
4.6.3 Impacts on Query Performance . . . 61
4.6.4 Efficiency of Fauce . . . 62
4.6.5 Handling Data Updates . . . 63
4.6.6 Other Factors Impacting Fauce . . . 67
4.6.7 Data Profiling . . . 68
4.7 Summary . . . 68

Chapter 5 Lobster: Load Balance-Aware I/O for Distributed DNN Training 70
5.1 Overview . . . 70
5.2 Motivation . . . 72
5.3 Design . . . 75
5.3.1 Flexible Preprocessing Thread Management . . . 76
5.3.2 Coordinated Data Loading / Preprocessing . . . 77
5.3.3 Performance Model . . . 78
5.3.4 Heuristic Strategy . . . 80
5.3.5 Implementation Details . . . 81
5.4 Evaluation . . . 82
5.4.1 Experimental Setup . . . 82
5.4.2 I/O Performance . . . 83
5.4.3 Reduction of Load Imbalance . . . 85
5.4.4 End-to-End Training . . . 86
5.4.5 Resource Utilization . . . 87
5.4.6 Ablation Study . . . 88
5.5 Summary . . . 88

Chapter 6 ArbiLIKE: An Accurate Cardinality Estimator for Arbitrary LIKE Predicates . . . 90
6.1 Overview of ArbiLIKE . . . 90
6.2 Problem Description . . . 91
6.3 LIKE Predicates Encoding . . . 93
6.3.1 Statistics Collection . . . 93
6.3.2 Cardinality-Distance Oriented Clustering . . . 94
6.3.3 Cluster-Centroid Embedding . . . 96
6.3.4 Cardinality-Aware Substrings Embedding . . . 99
6.4 Sequence model-based estimator . . . 100
6.4.1 Cardinality Estimation via Sequence Model . . . 100
6.4.2 Substring-importance Boosted Model . . . 101
6.5 Extension to Generic LIKE Predicates . . . 104
6.5.1 Formulation as a Set Resemblance Problem . . . 104
6.5.2 Signature Vector of Inverted List . . . 105
6.5.3 Extend to Multiple Columns . . . 107
6.6 Evaluation . . . 108
6.6.1 Experimental Setup . . . 109
6.6.2 Estimation Quality . . . 112
6.6.3 Estimations on Standard Benchmarks . . . 113
6.6.4 Impacts on Query Performance . . . 114
6.6.5 Hyper-parameter Tuning . . . 118
6.6.6 Efficiency of ArbiLIKE . . . 121
6.6.7 Handling String Indexing . . . 123
6.6.8 Handling Data Updates . . . 124
6.6.9 Impact of Embedding Methods . . . 126
6.6.10 Ablation Study of ArbiLIKE . . . 127
6.6.11 Varying Number of Wildcards . . . 128
6.7 Summary . . . 129

Chapter 7 Related Work . . . 130

Chapter 8 Conclusion and Future Work . . . 135

Bibliography . . . 138

LIST OF FIGURES


Figure 2.1: An example of heterogeneous mobile processors. . . . . 11
Figure 2.2: DNN training pipeline. . . . . 12
Figure 2.3: The storage hierarchy for distributed training in our environment. 13

Figure 3.1: An overview of the model design in Flame. . . . . 16
Figure 3.2: DNNs training pipeline. . . . . 18
Figure 3.3: Breakdown analysis of different components in Flame. . . . . 24
Figure 3.4: The interaction between CPU and DSP. . . . . 28
Figure 3.5: Comparison between different labeling methods in execution time. 33
Figure 3.6: Comparison of the energy consumption. . . . . 33
Figure 3.7: The execution time of the three versions of Flame. . . . . 34
Figure 3.8: Energy consumption comparison of different labeling methods. . 35
Figure 3.9: PassMark slowdown with auto-labeling running in background. . 37
Figure 3.10: Impacts of using Flame on user experience. . . . . 39
Figure 3.11: The frame latency distribution of using different versions of Flame. 39

Figure 4.1: Overview of Fauce. . . . . 42
Figure 4.2: An example of *Joins2Vec*. . . . . 45
Figure 4.3: End-to-end example of *Columns2Vec*. . . . . 47
Figure 4.4: Training and inference process of Fauce. . . . . 48
Figure 4.5: Integration of Fauce with existing DBMS . . . . . 55
Figure 4.6: Estimation errors on various datasets with no joins. . . . . 61
Figure 4.7: The impact of the improved cardinality estimation of Fauce on query performance. . . . . 61
Figure 4.8: Physical efficiency of Fauce. . . . . 62
Figure 4.9: Q-error of queries with different uncertainties. . . . . 63
Figure 4.10: Statistical information for the queries. . . . . 64
Figure 4.11: Estimation quality under dynamic environment. . . . . 65
Figure 4.12: Runtime for data profiling. "x" means out of limit. . . . . 69

Figure 5.1: Overview of Lobster. . . . . 71
Figure 5.2: Execution time breakdown for the training pipeline on three GPUs, two on one node and the third on a second node. . . . . 72
Figure 5.3: Histogram of the reuse distance of the training samples, measured in terms of numbers of iterations (X-axis) . . . . . 74
Figure 5.4: The impact of number of preprocessing threads (X-axis) on data preprocessing throughput (Y-axis) . . . . . 76
Figure 5.5: Comparison between Lobster and the baselines for multi-GPU training on a single node and distributed training across multiple nodes. . . . . 84
Figure 5.6: The number of iterations with load imbalance and the distribution of batch time. We use ResNet50 with ImageNet-1K. . . . . 85

Figure 5.7: Training accuracy curve for training ResNet50 on ImageNet-1K using eight nodes (64 GPUs) with default ResNet50 hyperparameter settings. . . . . 86
Figure 5.8: GPU utilization when training ResNet50 on ImageNet-1K using one node (eight GPUs). X-axis represents different DNNs used for testing and Y-axis is the GPU utilization. . . . . 87
Figure 5.9: Ablation study of Lobster when training ResNet50 on ImageNet-1K using one node (eight GPUs). Y-axis is the training time speedup compared with DALI. . . . . 88

Figure 6.1: ArbiLIKE overview. . . . . 91
Figure 6.2: $q$-grams and inverted lists. (a) The Customer table; (b) All the $q$-grams for the "Name" attribute ($q = 2$); (c) Each row consists of a 2-gram and corresponding inverted list. . . . . 92
Figure 6.3: The cardinality-aware substring embeddings involves three steps. Each "node" is a pair of (substring, cardinality). . . . . 94
Figure 6.4: The intra-cluster contrastive embedding for the pairs of (substring, cardinality) within a cluster. . . . . 100
Figure 6.5: Cardinality estimator in ArbiLIKE. . . . . 101
Figure 6.6: Illustration of generating the signature vectors of the inverted lists. (a) Inverted lists of $\mathbf{I}$(bi) and $\mathbf{I}$(go). (b) The permutations of the string IDs. The first row is the string IDs, and the first column represents four random permutations of the string IDs. (c) The signature vectors of $\mathbf{I}$(bi) and $\mathbf{I}$(go). . . . . 105
Figure 6.7: Cardinality estimation errors for the LIKE predicates within JOB and JOB-extend. The scale of the y-axis is logarithmic with base 10. . . . . 113
Figure 6.8: The percentage of queries completed throughout the runtime for both our workloads (Table 6.3) and the Join Order Benchmark (JOB). . . . . 115
Figure 6.9: Relative runtime improvement for the JOB, where each bar represents a single query. These queries have been sorted from best to worst improvement. The scale of the y-axis is logarithmic with base 10. . . . . 115
Figure 6.10: Relative runtime improvement for queries with LIKE predicates in JOB-extend, where each LIKE predicate is constructed using one or multiple string columns from the same table. . . . . 117
Figure 6.11: Relative runtime improvement for queries with LIKE predicates in JOB-extend, where each LIKE predicate is constructed using two or more string columns. . . . . 118
Figure 6.12: The impact of length of $q$-grams in ArbiLIKE (x-axis) on Q-errors (y-axis). . . . . 120
Figure 6.13: Signature vector size and storage space of ArbiLIKE. . . . . 121
Figure 6.14: Physical efficiency of ArbiLIKE. . . . . 123

Figure 6.15: Mean absolute errors for the string indexing of different methods. x-axis is the name of each datasets, y-axis is the indexing error. . 124
Figure 6.16: Estimation quality under dynamic environment. . . . . 125
Figure 6.17: The impact of different number of wildcards (x-axis) on Q-errors for LIKE predicates (y-axis). . . . . 127

LIST OF TABLES


Table 3.1: Characteristics of datasets. . . . . 30
Table 3.2: Comparison of labeling accuracy of different methods. . . . . 32
Table 3.3: Accuracy comparison between baselines and Flame. . . . . 33
Table 3.4: Ablation study results of different components in Flame. . . . . 35

Table 4.1: Query Features Segmentation . . . . . 48
Table 4.2: Workloads used for evaluation. . . . . 57
Table 4.3: Estimation errors on the JOB-base, JOB-more-filters, JOB-complex-joins workloads. . . . . 58
Table 4.4: Impact of encoding methods. . . . . 67
Table 4.5: Impact of cardinality transformation. . . . . 67
Table 4.6: Datasets used for data profiling. . . . . 69

Table 5.1: Notation used in performance models. . . . . 78

Table 6.1: Notations. . . . . 92
Table 6.2: Statistics for the datasets used in ArbiLIKE. . . . . 109
Table 6.3: Workloads used for evaluation. #Subs: Number of substrings. Wilds: Wildcards type contained in LIKE predicates. #Wilds: Number of wildcards in the workloads. Feature: Characteristic of LIKE predicates for each workload. . . . . 109
Table 6.4: Estimation errors on the three group of LIKE predicates workloads over five different datasets. . . . . 110
Table 6.5: Hyper-parameters considered for tuning. . . . . 119
Table 6.6: This table shows the impact of substring embedding size on the estimation results (§6.3.4). The result shows the 90th error across five datasets on QS-base workload. The lowest errors are bolded. . 120
Table 6.7: This table shows the impact of different encoding methods for LIKE predicates(§6.3.4). "PST" represents the encoding method based on prefix and suffix trees in Astrid. "Ours" denotes our encoding method for substrings. The lowest errors are bolded. . . . . 126
Table 6.8: Ablation studies: varying primary components of ArbiLIKE. We show the impact of (A) LIKE predicates encoding method (§6.3), (B)learned estimator (§6.4), and (C) set resemblance for arbitrary LIKE predicates (§6.5) on QS-multi-subs and QS-diff-wilds workloads over IMDB_AN dataset. . . . . 128

# ACKNOWLEDGEMENTS

I am grateful for the support from many great people during this long adventure. First of all, I would like to sincerely thank my advisor Professor Dong Li. His continuous guidance helped me overcome difficulties and achieved my goals throughout my graduate study, and inspired me to move towards a better career path.

I want to express my gratitude to my committee, Professor Florin Rusu and Professor Pengfei Su for the time and effort they spent to help me prepare my dissertation and serve as my committee members. I want to acknowledge Dr.Bogdan Nicolae, Dr. Min Si, and Professor Xia Ning for their help and guidance in the papers that we co-authored.

I appreciate my internship days at MemVerge, Tencent America, Argonne National Laboratory and Futurewei technology. My thanks go to Qingqing Zhou, Dr. Yue Li, and Dr. Jingchao Sun for much help along the way. I could never have imagined more enjoyable internship experiences.

I also would like to express my appreciation to colleagues and friends at UC Merced. I would like to thank Dr.Jiawen Liu and Prof.Zhen Xie for helping me in multiple projects and offering me with valuable suggestions. I am also grateful to other members of Parallel Architecture, System, and Algorithm Laboratory: Dr.Kai Wu, Prof.Wenqian Dong, Prof.Jie Ren, Mr. Dong Xu, Mr. Bin Ma, Mr. Jianbo Wu, and Ms. Sherry Wang. I give my heartfelt thanks my friends and roommates, with whom I had a wonderful time: Dr. Yuxin Tian, Dr. Yuanran Zhu, Mr.Yuning Chen, and Dr.Hao Xiong.

Last but not least I want to express my gratitude to my family for their love and support during my whole life. They giving me unconditional love and belief in me in every choice that I have made in my life.

ABSTRACT OF THE DISSERTATION

**Data-driven Performance Optimization for Data-intensive Applications**

by

Jie Liu

Doctor of Philosophy in Electrical Engineering and Computer Science

University of California Merced, Spring 2024

Dong Li, Chair

Data-intensive applications have attracted considerable attention from researchers in information sciences and enterprises, as these applications have made evolutionary breakthroughs in scientific fields and are extremely valuable to produce productivity in businesses. Recently, as the high speed growth of the new generated data, researchers have begun to leverage the useful knowledge hidden in such huge volume of data to optimize the performance of the data-intensive applications. However, optimize the performance of the data-intensive applications based on the data-driven approaches are still need to be explored.

In this thesis, we focus on data-driven performance optimization for data-intensive applications. We first study an application, auto-labeling data on mobile devices. How to accurately and efficiently label data on a mobile device is critical for the success of training machine learning models on mobile devices. Auto-labeling data for data-intensive applications on mobile devices is a challenging task, because data is incrementally generated and there is a possibility of having unknown labels among new coming data. Furthermore, the rich hardware heterogeneity on mobile devices creates challenges on efficiently executing the autolabeling workload. We introduce Flame, an auto-labeling system that can label dynamically generated data with unknown labels. Flame includes an execution engine that efficiently schedules and executes auto-labeling workloads on heterogeneous mobile processors. Evaluating Flame with six datasets on two mobile devices, we demonstrate that the labeling accuracy of Flame is 11.8%, 16.1%, 18.5%, and 25.2% higher than a state-of-the-art labeling method, transfer learning, semi-supervised learning, and boosting methods

respectively. Flame is also energy efficient, it consumes only 328.65mJ and 414.84mJ when labeling 500 data instances on Samsung S9 and Google Pixel2 respectively. Furthermore, running Flame on mobile devices only brings about 0.75 ms additional frame latency which is imperceivable by the users.

Second, we explore another data-intensive application, the cardinality estimation in database systems. Cardinality estimation is a fundamental and critical problem in databases. Recently, many estimators based on deep learning have been proposed to solve this problem and they have achieved promising results. However, these estimators struggle to provide accurate results for complex queries, due to not capturing real inter-column and inter-table correlations. Furthermore, none of these estimators contain the uncertainty information about their estimations. In this paper, we present a join cardinality estimator called . learns the correlations across all columns and all tables in the database. It also contains the uncertainty information of each estimation. Among all studied learned estimators, our results are promising: (1) has the smallest model size; (2) It has the fastest inference speed; (3) Compared with the state of the art estimator, has $10\times$ faster inference speed, and provides $1.3\times \sim 6.7\times$ smaller estimation errors for complex queries; (4) To the best of our knowledge, is the first estimator that incorporates uncertainty information for cardinality estimation into a deep learning model.

Furthermore, we also study the data loading problem for large-scale distributed training. The resource-hungry and time-consuming process of training Deep Neural Networks (DNNs) can be accelerated by optimizing and/or scaling computations on accelerators such as GPUs. However, the loading and pre-processing of training samples then often emerges as a new bottleneck. This data loading process engages a complex pipeline that extends from the sampling of training data on external storage to delivery of those data to GPUs, and that comprises not only expensive I/O operations but also decoding, shuffling, batching, augmentation, and other operations. We propose in this paper a new holistic approach to data loading that addresses three challenges not sufficiently addressed by other methods: I/O load imbalances among the GPUs on a node; rigid resource allocations to data loading and data preprocessing steps, which lead to idle resources and bottlenecks; and limited efficiency of caching strategies based on pre-fetching due to eviction of training samples needed soon at the expense of those needed later. We first present a study of key bot-

tlenecks observed as training samples flow through the data loading and preprocessing pipeline. Then, we describe Lobster, a data loading runtime that uses performance modeling and advanced heuristics to combine flexible thread management with optimized eviction for distributed caching in order to mitigate I/O overheads and load imbalances. Experiments with a range of models and datasets show that the Lobster approach reduces both I/O overheads and end-to-end training times by up to 1.5× compared with state-of-the-art approaches.

Finally, we study the cardinality estimation for string predicates in database systems. Cardinality estimation for string predicates is a notoriously challenging problem in database systems. This paper presents ArbiLIKE, an advanced deep learning-based cardinality estimator for arbitrary LIKE predicates. ArbiLIKE utilizes a cardinality-aware embedding technique to encode LIKE predicates into feature vectors. It further incorporates an innovative sequence model to capture the semantic information of different substrings, enhancing the estimation accuracy. ArbiLIKE is also capable of handling LIKE predicates with any combination of wildcards ("%", "_"). Empirical evaluations showcase ArbiLIKE's promising accuracy, achieving estimation errors that are up to 165.1× smaller than those of eight baselines, including state-of-the-art methods. As a generic estimator, ArbiLIKE realizes error reductions ranging from 1.4 to 93.1× for LIKE predicates with multiple wildcards in comparison to the existing techniques. To the best of our knowledge, ArbiLIKE is the first deep learning-based estimator capable of handling arbitrary LIKE predicates.

# Chapter 1

# Introduction

With the huge volume of generated data, data-intensive applications have drawn huge attention from both academic and industry. Those data-intensive applications increasingly face performance problems, because of hardware heterogeneity, difficulty of extracting valuable information from dynamic generated data, and extraordinary techniques to efficiently process large volume of data within limited run times. To optimize the performance of data-intensive applications, it requires the researchers to propose more sophisticated approaches.

Some applications running on mobile devices are data-intensive and their performance needs to be optimized. For example, when training RNN model on mobile devices to recommend the appearance of the next words when use the keyboard for the words typing, it requires amount of labeled data. While the data labeling task on mobile devices can be very data-intensive. From the hardware perspective, the modern mobile processors in mobile devices are characterized with hardware heterogeneity. For example, mobile devices like Samsung S9 and Google Pixel2 have eight-core CPU (four slow cores and four fast cores), mobile GPU, and DSP. DSP is typically the most power-efficient computing unit in a mobile device [1]. To optimize the performance for data-intensive applications on mobile devices, we need to leverage the hardware heterogeneity to propose some sophisticated methods.

Furthermore, some database related applications are also data-intensive, for example, the cardinality estimation usually involves huge volume of data in the database systems. Cardinality estimation is fundamental and critical in databases. It is widely applied to query optimization, query processing approximation, database tuning, etc.

For example, the query optimizer uses the results of the cardinality estimation to determine the best execution plans. We study both the cardinality estimation for numeric-based and string-based queries. As string-based data becomes increasingly prevalent in relational databases, there is a growing need of accurate cardinality estimation for string queries. In this paper, we focus on cardinality estimations for queries involving string predicates, such as the SQL LIKE operator.

Finally, large-scale distributed DNNs training is a well known data intensive applications. Deep Neural Networks (DNNs) are rapidly gaining traction in both industry and scientific computing, driven by the accumulation of massive data. In science, for example, instruments that collect data at GB/s and 100+ TB/day present a wide range of learning opportunities. We thus see significant interest in deploying DNNs on high-performance computing (HPC) systems in order to enable rapid learning in domains such as computational fluid dynamics [2], power grids [3], and molecular dynamics [4]. Various approaches [5, 6] for training DL models on massive data have been developed: coarse-grain parallelization on multiple nodes using data-parallel, model-parallel, pipeline-parallel, and hybrid techniques; fine-grain parallelization on many-core architectures by constructing and scheduling execution graphs at the tensor level; and low-level optimizations of operators [7] and communication primitives [8].

However, the current methodology to optimize the aforementioned data-intensive applications faces the following challenges.

**Resource Constraints and Efficiency**. A significant challenge in deploying advanced functionalities on mobile devices and during large-scale distributed training is managing limited computational resources and memory efficiently. For instance, in mobile data labeling, methods like those discussed in [9, 10] require a large number of labeling functions and substantial computational resources, often exceeding the capacity of mobile devices. These devices, exemplified by the Samsung S9 with its mix of CPU cores, a GPU, and a Hexagon DSP, are constrained in terms of memory and processing power, limiting their ability to handle data-intensive tasks. Similarly, in the context of distributed deep neural network (DNN) training, there's a challenge in balancing load across GPUs. Variations in data fetch times due to differences in data storage locations (local cache vs. remote storage) can create bottlenecks, affecting overall training efficiency. Efficiently managing these resources is crucial for enhancing performance without necessitating hardware upgrades.

**Dynamic and Complex Data Handling**. Dealing with dynamically generated data and complex data patterns presents a substantial challenge in various technological applications. For example, in mobile applications, data is often generated in real-time, such as images for classification. This dynamic generation can result in new, unseen labels (e.g., new types of flowers) that the system must recognize and classify without prior knowledge. This scenario demands adaptive algorithms capable of handling new data categories as they appear. In the realm of database management, particularly in cardinality estimation for queries involving string-based predicates (e.g., LIKE predicates), the complexity escalates. Traditional embedding techniques fall short in accurately capturing the nuanced patterns of string-based data, necessitating innovative approaches to comprehend the intricacies of these queries, such as differentiating between "LIKE %abc%" and "LIKE %xyz%" based on their actual cardinalities.

**Heterogeneity and Coordination**. The diversity of hardware components and the need for their coordinated operation pose a significant challenge, especially in mobile computing and distributed systems. Mobile devices often contain a heterogeneous mix of processors, each with unique performance characteristics and energy efficiency levels. Optimally scheduling tasks across these varied components requires a deep understanding of their operational nuances to maximize performance and energy efficiency. In distributed DNN training, this challenge is mirrored in the coordination of data loading processes across multiple GPUs. Ensuring that all GPUs receive data at a similar pace to prevent stragglers requires sophisticated scheduling and resource management strategies. Moreover, caching mechanisms must be intelligently designed to pre-emptively load data in a manner that minimizes I/O overheads without causing premature cache evictions, which could otherwise impede training efficiency.

These challenges underscore the need for advanced, efficient, and adaptive solutions capable of navigating the complexities of resource constraints, dynamic data, and heterogeneous computing environments. Examples from the original text illustrate the practical implications of these challenges in real-world settings, highlighting the necessity for ongoing innovation in these areas.

## 1.1 Primary Contributions

This dissertation tries to resolve the above three challenges. It includes optimizing the performance for data-intensive applications based on the data-driven approaches(Chapters 2, 3, 4, and 5).

**Data labeling on heterogeneous mobile processors.** To address the above challenges, we introduce Flame, an auto-labeling system for mobile processors. Flame is featured with mobile hardware-aware algorithms and system designs.

To overcome the hardware resource constraint, Flame includes a new lightweight method, named *clustering with minimal impurity*, to build a number of labeling functions. After the clusters are built based on a limited number of labeled data instances, Flame replaces the data instances within the same cluster by the cluster's *prototypes* to reduce the computation overhead. The decision boundary of each labeling function is determined by its prototypes. Any data instance falling outside the decision boundaries of all the labeling functions is identified as a data instance potentially with a new label. Flame interprets the presence of a sufficiently large number of such data instances with strong cohesion among themselves as the emergence of a new label. Furthermore, because of the dynamic characteristics of the data to be labeled on mobile devices, the labeling functions must be updated from time to time to capture the accurate distribution of data. In Flame, because each labeling function consists of a number of prototypes, and updating the labeling functions is just a matter of updating its prototypes.

To guarantee the labeling accuracy, Flame uses two estimators, *Association* and *Purity*, to measure the labeling confidence of each labeling function. We theoretically show that the use of these estimators can guarantee the labeling accuracy. Finally, Flame uses an ensemble method to gather the labeling confidence of labeling functions to determine final labels.

Flame is featured with a hardware heterogeneity-aware execution engine to run the auto-labeling algorithm (§3.3). The execution engine determines which part of the computation should be placed on a particular computing unit (CPU, GPU or DSP) based on the characteristics of workload and hardware. Some computation of Flame is placed on GPU to shorten execution time, because the computation can offer high thread-level parallelism and efficiently leverage fast shared-memory on GPU. Some

computation is placed on DSP (the most power-efficient computing unit), when the energy consumption of the computation is high. The execution engine also coordinates the interaction between CPU and DSP to avoid wakeup latency suffered by CPU for energy saving. We summarize major contributions as follows.

The labeling accuracy of Flame is 11.8%, 16.1%, 18.5%, and 25.2% higher than that of Snuba [10] (a state-of-the-art auto-labeling system), transfer learning, semi-supervised learning algorithms and boosting methods respectively. Also, Flame can detect unseen labels while all other systems cannot. Flame has high energy efficiency. It consumes only 328.65mJ and 414.84 mJ when labeling 500 data instances on Samsung S9 and Google Pixel2 respectively, while the full energy of their battery is $3.88 \times 10^4$ and $3.49 \times 10^4$J respectively. This makes Flame a highly feasible system for mobile phones. Flame running on a mobile phone has minimum impacts on the user experience of using another application. Flame brings only 0.75 ms additional frame latency to the user application, which is ignorable, compared with the minimum threshold of user-perceivable latency (100 ms [11]).

**Cardinality estimation for numberic predicates with uncertainty.** We propose a new cardinality estimator, Fauce. Fauce includes a new query featurization method (§4.3) that leverages semantic information contained in the database and captures real dependent relationships between table columns to encode the queries into more informative feature vectors. Furthermore, we mathematically define the uncertainty of the estimator and introduce a new model that incorporates the uncertainty estimation into Fauce (§6.4). Fauce also includes a new learning paradigm that leverages the uncertainties to boost the estimation results and make Fauce robust to be applied in dynamic databases (§4.5.3).

To capture the real correlations across all the table columns in a database (§4.3.2), we introduce dependency graphs to capture dependent relationships across columns, and based on the graphs we embed the columns into a vector to boost the estimation accuracy.Using a general structure to capture dependency requires the catch of implicit dependency relationships in columns across tables. Catching such a dependency is difficult [12, 13, 14]. Our dependency graph is hierarchical. In particular, we first build a local columns-dependency graph for each table. Then we build the global columns-dependency graph for all the columns in the database based on the local columns-dependency graphs developed in the first step. Finally, we use an em-

bedding technique [15] to represent each column into a vector based on the global columns-dependency graph. Such vectors can convey real correlations among the columns.

To include the uncertainties of the cardinality estimator into Fauce, we design a model based on *deep ensembles* (§4.5.3) to comprehensively quantify the uncertainty. The uncertainty of the cardinality estimator comes from multiple sources. First, we are uncertain about whether the learned model parameters can best describe the distribution of the queries in the query space. This is referred to as *model uncertainty*. Second, the query-based estimators train the model based on the generated training dataset. But the training dataset can not well reflect the features for all the queries. That is to say, there is always a data shift between the training dataset and the inference queries. This data shift can be large especially for dynamic databases. Thus, we are also uncertain about whether the data used to train model can well represent the features for inference queries, this is referred as *data uncertainty*. These two types of uncertainty consist the uncertainty of the learned estimator.

We conducted an extensive set of experiments over IMDB, a real-world dataset that exhibits complex correlation and conditional independence between table columns and have been extensively used in prior work [16, 17, 18, 19, 20]. On the created JOB-base benchmark, a schema that contains 6 tables and basic filters. Fauce achieves 1.16-4× higher accuracy over the state of the art estimator, it also consumes 1.2× less memory footprint and over 5× faster for inference. To check whether Fauce is robust to complicated queries with large number of filters, we create a more difficult benchmark, JOB-filters-extension. On this benchmark, Fauce achieves up to 1.16-51× higher accuracy than previous estimators, including IBJS [21], MSCN [16], DeepDB [17],and NeuroCard [19]. Lastly, to test Fauce's ability to handle queries with more complex join relations, we created JOB-joins-extension which has 15 tables and complex joins. Experimental results show that Fauce scales well to this benchmark, it has up to 1.16-91× higher accuracy than baselines.

**Load balance-aware I/O for distributed DNNs training.** We propose Lobster, a holistic data loading I/O runtime for distributed DNN training. Lobster distinguishes between the I/O load of each individual GPU at fine granularity and coordinates the I/O operations of the GPUs at the node level, flexibly allocating available I/O bandwidth and threads as needed to reduce I/O load imbalance.

We characterize the performance (especially I/O performance) across 64 GPUs in a production environment for distributed DNN training, highlighting the I/O load imbalance across GPUs and frequent performance bottleneck shifts between data loading/pre-processing pipeline and the training process. This study reveals new opportunities for I/O performance optimization that are not considered by state-of-art approaches.

We propose a thread management strategy to coordinate the resource usage between data loading and preprocessing in the training pipeline, as well as to mitigate the I/O load imbalance between GPUs. We introduce a holistic performance model that bridges the thread management strategy with a distributed caching proposal that features prefetching support and optimized eviction based on reuse distance. We also design and implement a heuristic strategy to solve the optimization problem resulting from the performance model. This strategy consists of two phases (prefetching and eviction) and guides both the allocation of the threads and the distributed caching.

Lobster also coordinates the data loading and preprocessing stages of the pipeline, flexibly allocating threads between them to reduce bottlenecks. This coordination is achieved through the use of performance modeling, which we combine with reuse distance theory to design efficient eviction policies for distributed caching of the training samples. Such an optimized eviction policy complements state-of-the-art distributed caching approaches based on prefetching by avoiding the undesirable effect of evicting training samples that are needed in the near future in order to make room for prefetched samples that are needed later. We show that this method increases the cache hit ratio by 14.3% compared with state-of-the-art prefetching approaches such as that used in NoPFS [22].

**Accurate cardinality estimator for arbitrary LIKE predicates.** We present ArbiLIKE, an advanced deep learning-based cardinality estimator for arbitrary LIKE predicates. ArbiLIKE utilizes a cardinality-aware embedding technique to encode LIKE predicates into feature vectors. It further incorporates an innovative sequence model to capture the semantic information of different substrings, enhancing the estimation accuracy. ArbiLIKE is also capable of handling LIKE predicates with any combination of wildcards (“%”, “_”).

To achieve cardinality-aware embeddings for LIKE predicates, ArbiLIKE introduces a *multi-tiered contrastive embedding* method. First, using the $q$-grams tech-

nique, ArbiLIKE identifies (substring, cardinality) pairs within the database. Subsequently, these pairs are structured hierarchically using a bottom-up clustering technique, leading to a dendrogram that distinctly captures the cardinality discrepancies across various substrings. Utilizing this structure, ArbiLIKE generates substring embeddings through two phases: the inter-cluster embedding phase, which encodes cluster centroids into feature vectors, and the intra-cluster embedding phase, where individual substrings are embedded, guided by the outcomes from inter-cluster embedding stage. Our proposed embedding methodology ensures that substrings with minor cardinality variations yield similar embeddings, whereas those with marked disparities are distinctly represented.

To derive a substring importance-boosted estimator for LIKE predicates, ArbiLIKE introduces a *substring importance-boosted sequence model*. This model captures both prefix and suffix patterns of characters, along with substring-level semantic information. Additionally, by integrating a *multi-dimensional self-attention mechanism* into the estimator, ArbiLIKE quantifies and utilizes the varying significance of different substrings during the cardinality estimation, leading to accuracy improvement of cardinality estimation.

ArbiLIKE supports arbitrary LIKE predicates by formulating the cardinality estimation for LIKE predicates with multiple wildcards ("%", "_") as a *set resemblance approximation* problem. In particular, ArbiLIKE decomposes the LIKE predicate into multiple LIKE sub-predicates, determined by the presence of wildcards. Subsequently, ArbiLIKE adopts a *Monte Carlo approximation* technique to efficiently calculate a resemblance value among the decomposed LIKE sub-predicates. Such a resemblance value, in conjunction with the estimated cardinality for each LIKE sub-predicate, is utilized to estimate the cardinality for arbitrary LIKE predicates.

To the best of our knowledge, ArbiLIKE is the first DL-based cardinality estimator for arbitrary LIKE predicates. ArbiLIKE leads the accuracy among the baselines we studied in the paper. We conduct an extensive set of experiments using real-world datasets, IMDB [23] and DBLP [24], which contain numerous string attributes. We evaluate ArbiLIKE from the accuracy and generalization perspectives. The experimental results demonstrate that ArbiLIKE achieves 1.46-96.4× higher accuracy than six baselines on arbitrary LIKE predicates, including Postgres [25], KVI [26], MO [27], BayesNet [28], CRT [29], and Astrid [30].

## 1.2 Outline and Previously Published Work

The remainder of the dissertation is organized as follows. Chapter 2 provides a comprehensive background for this dissertation. Chapter 3 presents the design, implementation and evaluation of Flame, a fast, accurate, and lightweight auto-labeling system for mobile devices. Chapter 4 introduces Fauce, the first learned cardinality estimator that contains the uncertainties for its results. Chapter 5 characterizes the performance (especially I/O performance) across 64 GPUs in a production environment for distributed DNN training. We describe Lobster, a data loading runtime that uses performance modeling and advanced heuristics to combine flexible thread management with optimized eviction for distributed caching in order to mitigate I/O overheads and load imbalances. Chapter 6 introduces ArbiLIKE (§6.3), the first DL-based cardinality estimator for arbitrary LIKE predicates, outperforming all studied baselines in accuracy. Chapter 7 includes the related work of this dissertation. Chapter 8 concludes this dissertation by summarizing the main lessons learned, the open problems, and the topics for future work.

Chapter 2 contains material of background section from several published papers[31, 32, 33], The dissertation author is the primary investigator and first author of these papers.

Chapter 3 contains material from "Flame: A Self-Adaptive Auto-Labeling System for Heterogeneous Mobile Processors", by by Jie Liu, Jiawen Liu, Zhen Xie, Xia Ning and Dong Li, which appears in the Proceedings of the ACM/IEEE Symposium on Edge Computing (SEC'21). The dissertation author is the primary investigator and first author of this paper. 

Chapter 4 contains material from "Fauce: Deep Ensembles with Uncertainty for Cardinality Estimation", by Jie Liu, Wenqian Dong, Qingqing Zhou, and Dong Li, which appears in the 47th International Conference on Very Large Data Bases (VLDB'21). The dissertation author is the primary investigator and first author of this paper. 

Chapter 5 contains material from "Lobster: Load Balance-Aware I/O for Distributed DNN Training", by Jie Liu, Bogdan Nicolae, and Dong Li, which appears

in the 51th International Conference on Parallel Processing (ICPP'22). The dissertation author is the primary investigator and first author of this paper. The material in these chapters is copyright ©2022 by the IEEE Association.

Chapter 6 contains material from "ArbiLIKE: An Accurate Cardinality Estimator for Arbitrary LIKE Predicates", by Jie Liu, Qingqing Zhou, and Dong Li, which is submitted in the 2025 ACM SIGMOD International Conference on Management of Data (SIGMOD'25). The dissertation author is the primary investigator of these papers. The material in thesis chapter is copyright ©2025 by the Association for Computing Machinery (ACM).

# Chapter 2

# Background

This chapter presents a comprehensive introduction to the background of this dissertation.

## 2.1 Background on Mobile Processors

We introduce background of Flame information in this section.

**Heterogeneous mobile processors.** The modern mobile processors in mobile devices are characterized with hardware heterogeneity. Figure 2.1 shows such an example commonly found in many mobile devices (e.g., Samsung S9, Google pixel2, Huawei P8 and Xiaomi Mi 10). In our study, we use mobile devices, each of which has eight-core CPU (four slow cores and four fast cores), mobile GPU, and DSP. DSP is typically the most power-efficient computing unit in a mobile device [1]. However, DSP is not good at handling some operations (e.g., square root and division).

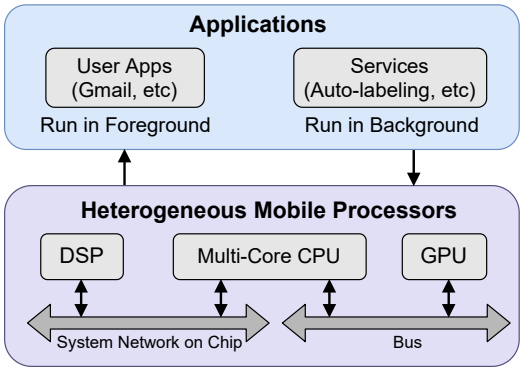


**Figure 2.1:** An example of heterogeneous mobile processors.

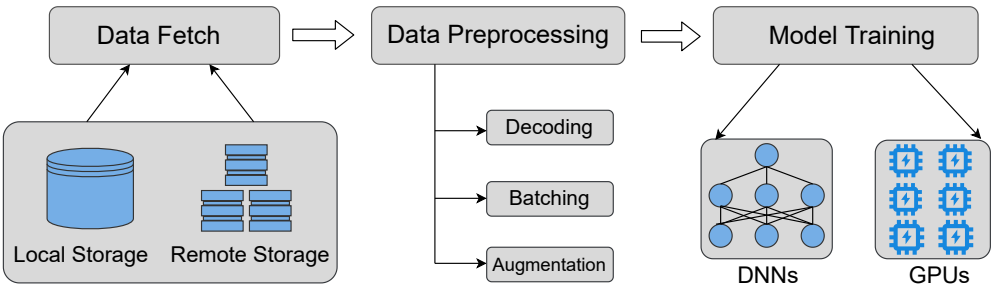


**Figure 2.2:** DNN training pipeline.

**Mobile applications.** The applications running on mobile devices can be classified as foreground and background applications. The mobile system usually sets a higher priority to run the foreground applications to enable smooth interaction between the user and mobile devices. The background applications have low priority to be scheduled and run by the mobile system [34]. The background applications are not expected to introduce significant latency to the foreground applications. Flame is a background application.

## 2.2 Background of DNNs Training

DNN training is an iterative process: first, the answer to an input is obtained in a forward pass over all layers. Then, in a backward pass, the difference (gradients) between the predicted and actual result ("ground truth") is used to update the weights layer by layer in reverse order. This process repeats for a large number of iterations until the DNN model has converged. Typically, multiple passes over the whole training data are required. Thus, iterations are grouped into epochs, each of which represents a full pass.

The input of each iteration is a *mini-batch*, which is obtained by random sampling of the training data. For efficiency reasons, in practice a pseudo-random number generator is used to shuffle the training samples, after which they are accessed in the shuffled order and grouped together as mini-batches. Since the seed of the pseudo-random number generator is known in advance, the I/O access pattern necessary to read the training samples can be made fully deterministic [22].

Before executing the forward and backward pass, the DNN training pipeline includes a data loading and preprocessing state, as illustrated in Figure 2.2. Data

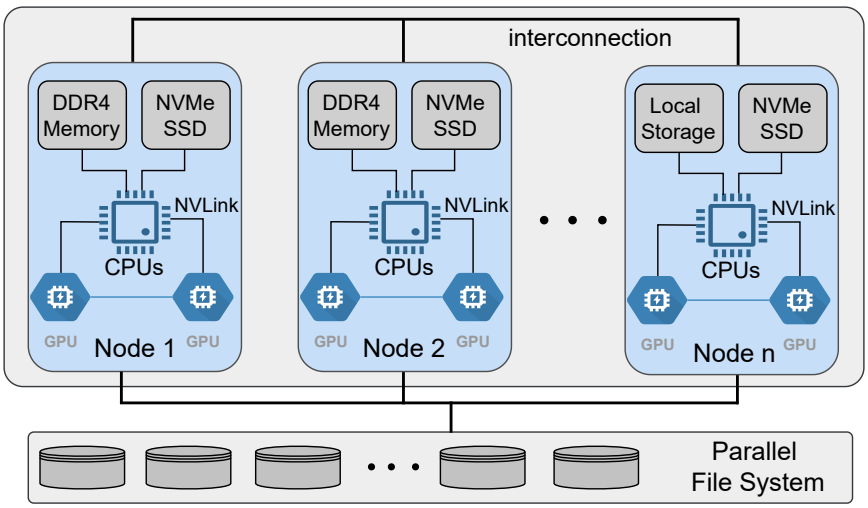


**Figure 2.3:** The storage hierarchy for distributed training in our environment.

loading is responsible for prefetching and caching the training samples (which is possible thanks to the I/O access pattern being deterministic), while data preprocessing is responsible for additional transformations: decoding, augmentation, batching. All these stages in the pipeline are overlapping, which optimizes the resource utilization. The data preprocessing can be performed on either the CPU or the GPUs. For the purpose of this work, we assume the preprocessing is performed on the CPU, while the training is performed on the GPUs. This is a common scenario [35, 36, 37, 38, 39, 40] that makes efficient use of heterogeneous compute resources.

In order to scale the DNN training, multiple nodes equipped with multiple GPUs are used, as illustrated in Figure 2.3. The most common approach to achieve this is *data parallelism*, i.e., training the same DNN model replica on multiple GPUs with different mini-batches, then averaging the gradients during the backward pass. In this case, the GPUs co-located on the same node share a node-local cache. If a training sample is not available in the node-local cache, it can be retrieved either from the external storage repository (typically a parallel file system or PFS) or, if available, from the cache of a different node. The latter requires the implementation of a distributed cache but improves I/O latency significantly for several reasons: (1) the bandwidth between compute nodes is higher than the I/O bandwidth between a single compute node and the PFS; (2) the aggregated I/O bandwidth of the PFS is limited and becomes a bottleneck when multiple compute nodes compete for it; (3) the PFS is not optimized for I/O access patterns that involve small randomly

scattered reads necessary to retrieve the training samples.

In Lobster, we assume a data-parallel training that makes use of a distributed cache. However, it is important to note that our proposal works in general for other DNN training scenarios as well (e.g., different DNN models sharing the same training data, alternatives to distributed caching like for example KV-stores, or even single-node DNN training). Our goal is to optimize node-local caches such that they can serve multiple co-located GPUs efficiently, when considering the challenges discussed: (1) data load imbalance across the GPUs; (2) lack of coordination between the stages of the pipeline; (3) sub-optimal cache eviction due to deterministic prefetching.

# Chapter 3

# Flame: A Self-Adaptive Auto-Labeling System for Heterogeneous Mobile Processors

## 3.1 Overview

We propose a fast, accurate, and lightweight auto-labeling system, Flame, for mobile devices. It is the first system to label data dynamically generated on mobile devices. We implement it on two realistic mobile phones (Samsung S9 and Google Pixel2). (1) Labeling Functions Generation generates a number of labeling functions (LFs) based on the user labeled data instances, each labeling function includes several prototypes. (2) Labeling Functions Self-adaption applies the LFs on the dynamic unlabeled data, these LFs can be updated by the Flame. Furthermore, Flame can detect the emergence of new labels and building new labeling functions for the data instances with new labels. (3) Labeling Results Guarantees calculates a labeling confidence value for each labeling function, then it aggregates all the labeling confidence values, finally, it assigns labels for unlabeled data instances. (4) The labeled data instances can be used to train discriminative classification models, such as a deep neural network.

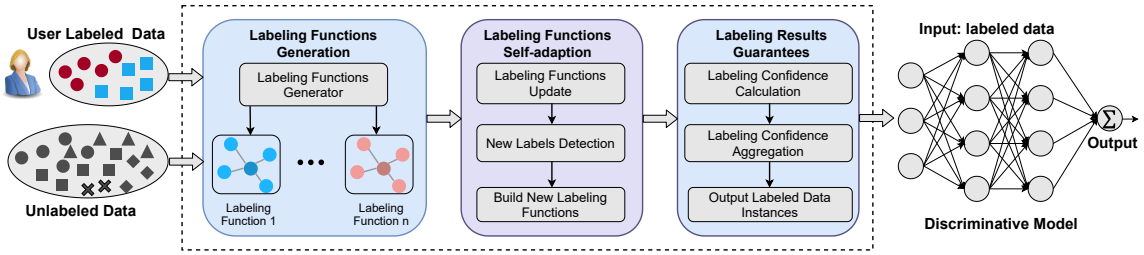


**Figure 3.1:** An overview of the model design in Flame.

## 3.2 Model Design

Figure 3.1 overviews our auto-labeling model. Flame includes three components, *Labeling Functions Generation*: a labeling function generator to generate a number of labeling functions for assigning labels. *Labeling Functions Self-adaption*: a self-adaptive strategy to update the existing labeling functions, detect the emergence of new unseen labels in a dynamic setting, and build new labeling functions for the data instances belong to new unseen labels. *Labeling Results Guarantees* calculates a labeling confidence value for each labeling function during data instance labeling, and then aggregates and verifies the labeling results of Flame for an unlabeled data instance. The input/output of Flame is discussed as follows.

**Input data.** The input data of Flame is a small number of labeled data and a large number of unlabeled data. The labeled data is represented as $D_L = \{x_i, y_i\}_{i=1}^{N_L}$, where $x_i \in R^d$ is the $d$-dimension features of the data $i$ and $y_i \in Y = \{1, 2, ..., C\}$ is the associated label ($C$ different known labels in total). The non-stationary unlabeled data is represented as $D_U = \{x_t\}_{t=1}^{N_U}$ ($x_t \in R^d$), where $N_U \in [0, \infty)$ is the number of unlabeled data. In our setting, $N_U$ can be large, as the new data is continuously generated.

**Output data.** The output of Flame is the confidence of a label $y_i \in Y' = \{1, 2, ..., C, ..., C'\}$ for data $x_i$ in the unlabeled dataset $D_U$, where $Y'$ is the set of result labels including known labels and new unseen labels ($C' \geq C$). Here, $C' \geq C$, which indicates that some unseen labels that are not in $Y$ may appear in $Y'$, as new data is incrementally generated. The final labeling confidence value is calculated through an ensemble method in Flame (§3.2.3).

### 3.2.1 Labeling Functions Generation

We design a lightweight method to generate labeling functions. Existing studies [9, 41] use supervised machine learning models (e.g., Decision Tree, K-Nearest Neighbor) to build labeling functions. However, these machine learning models [42, 43, 44, 45] cannot work well on mobile devices because of two reasons. First, the data generated on mobile devices are seldom labeled. Therefore, using the labeling functions built based on supervised learning models causes low labeling accuracy. Second, a large number of labeling functions (more than 100) are needed to have high data coverage, which requires abundant computational resources and memory space.

We design an impurity-based clustering method to determine the boundary of each labeling function. A cluster is completely *pure* if the data instances within this cluster belong to the same label (along with some unlabeled data). Given a limited amount of labeled data, the goal of impurity-based clustering is to create a number of clusters by minimizing the intra-cluster dispersion, and at the same time by minimizing the impurity of each cluster, we refer it as *Clustering with Minimal Impurity*. In order to determine the boundary of each labeling function in fine granularity, each created cluster is further divided into a number of cliques. A clique of a cluster is a group of data instances with cohesion larger than 0.5 in the corresponding cluster, the cohesion is calculated by a commonly used method called q-NSC [46].

Then, we use the *prototype* of a clique to replace the data instances in that clique. A prototype indicates the best exemplar of the data instances within a clique. The corresponding prototypes of all the cliques of a cluster could provide a concise representation for the entire raw data instances within a cluster. In this paper, a prototype of a clique is defined as below,

**Prototype**: the prototype of a clique is a tuple denoted by $p =< \mu, r, d, n, \bar{f} >$, where $\mu$ is the centroid of the clique, $r$ represents the radius of the clique, $d$ denotes the sum of squared Euclidean distance from data instances in the clique to $\mu$, $n$ is the total number of data instances in the clique, and $\bar{f}$ is a vector recording the number of data instances belonging to different labels in the clique.

The $\bar{f}$ is referred as *frequencies* in the rest of the paper. Here is an example of $\bar{f}$, $\bar{f} = (f_1, f_2, ..., f_t)$, where each element $f_i$ in $\bar{f}$ is the frequency of the corresponding label $y_i$ assigned to the data instances. Finally, each cluster is represented by one or

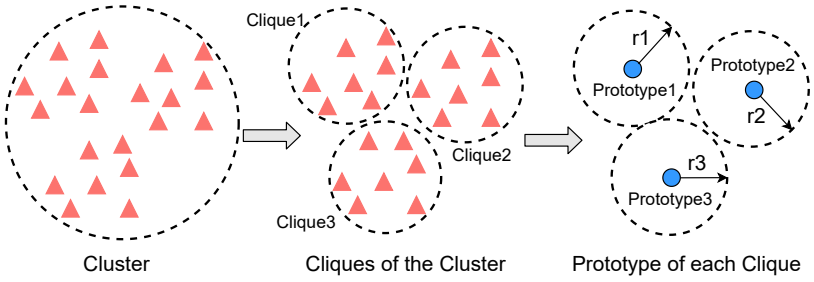


**Figure 3.2:** DNNs training pipeline.

**Algorithm 1** PrototypeInitialization ($D_{ij}$)

**Input** : $D_{ij}$: the data instances in clique $C_{ij}$ of cluster $i$
**Output:** $p_{ij}$: the initialized prototype for clique $C_{ij}$
$\mu_{ij} = \frac{1}{|D_{ij}|}\sum_{x \in D_{ij}} x$; // $|D_{ij}|$ `is the size of` $D_{ij}$
$r_{ij} \leftarrow \max_{x \in D_{ij}} \|x - \mu_{ij}\|_2^2$ $d_{ij} \leftarrow \sum_{x \in D_{ij}} \|x - \mu_{ij}\|_2^2$ $n_{ij} \leftarrow |D_{ij}|$ $\bar{f} \leftarrow (f_1, ..., f_i + 1, ..., f_C)$ **return** $p_{ij}$

more prototypes depending on the data distribution in the cluster. Each prototype is denoted by $p_{ij}$, where the subscript $i$ indicates the index of the $i^{th}$ built cluster, $j > 0$ is the prototype index. We denote the set of prototypes for cluster $i$ by $P_i$, i.e., $p_{ij} \in P_i$. Figure 3.2 depicts an example of the relationship among cluster, cliques, and prototypes.

The boundary of each labeling function is determined by a collection of one or multiple prototypes. Flame maintains a labeling function pool which is represented as $LF$, assuming $LF$ contains $K$ labeling functions, that is, $LF = \{lf_i, ..., lf_K\}$, where $lf_i \in LF$ is an individual labeling function. Each labeling function $lf_i$ could consisted by $M$ prototypes, therefore, $lf_i$ in $LF$ is represented as $lf_i = \{p_{i1}, ..., p_{iM}\}$.

**Prototype initialization.** The initialization for the prototypes is based on the limited number of labeled dataset $D_L = \{x_i, y_i\}_{i=1}^{N_L}$. Here, the number of labels in $D_L$ may be small when compared to the eventual labels that may occur over time. Data instances associated with label $y_i \in Y$ are denoted by $D_i$. $D_i$ consists of $M$ cliques $(C_{i1}, ..., C_{iM})$ and the data instances in clique $C_{ij}$ are represented as $D_{ij}$. We create a prototype for each clique by selecting a data instance from $D_{ij}$, uniformly at random. Algorithm 1 details the prototype initialization process for a given label $y_i \in Y$.

**Objective function.** When building the labeling functions using the *Clustering with Minimal Impurity* method, the objective is to minimize the dispersion and

impurity of clusters. We formulate the objective function as follows,

$$Obj(x) = \sum_{i=1}^{K} \sum_{j=1}^{M} \sum_{x \in D_{ij}} \|x - \mu_{ij}\|^2 + \lambda \sum_{i=1}^{K} \sum_{j=1}^{M} ADC_{ij} \times Entropy_{ij} \quad (3.1)$$

In Equation 3.1, the first term is used to minimize the dispersion of data instances within the scope of each prototype; $K$ is the total number of labeling functions in $LF$; $D_{ij}$ is the set of data instances within the scope of the prototype $p_{ij}$; and $\mu_{ij}$ is the centroid of the prototype $p_{ij}$. The second term in Equation 3.1 is used to minimize the impurity of data instances in each clique, and $\lambda$ is a hyper-parameter controlling the importance of the second term. The impurity is constructed based on labeling diversity and the entropy value of data instances in the scope of a prototype, and it is calculated as $ADC_{ij} \times Entropy_{ij}$, where $ADC_{ij}$ is the *Aggregated Dissimilarity Count* (ADC) of the prototype $p_{ij}$ and $Entroty_{ij}$ is the entropy of the prototype $p_{ij}$. $ADC_{ij}$ is calculated as follows,

$$ADC_{ij} = \sum_{x \in D_{ij}} \sum_{j=1}^{M} DC_{ij}(x, \ell), \quad (3.2)$$

where $DC_{ij}(x, \ell)$ denotes the *Dissimilarity Count* (DC) of a data instance $x$ in the prototype $p_{ij}$ having the label $\ell$, it is calculated as the total number of labeled instances in that prototype belonging to labels other than $\ell$.That is,

$$DC_{ij}(x, \ell) = |L_{ij}| - |L_{ij}(\ell)|, \quad (3.3)$$

where $|L_{ij}|$ is the total number of labeled data instances within the scope of prototype $p_{ij}$, and $|L_{ij}(\ell)|$ is the number of instances in the prototype $p_{ij}$ belonging to label $\ell$. The entropy in Equation 3.1 is calculated as follows,

$$Entropy_{ij} = \sum_{\ell=1}^{C'} (-\frac{|L_{ij}(\ell)|}{|L_{ij}|} \times \log(\frac{|L_{ij}(\ell)|}{|L_{ij}|})), \quad (3.4)$$

where $\frac{|L_{ij}(\ell)|}{|L_{ij}|}$ is the prior probability of the label $\ell$, $C'$ is the number of labels.

**Algorithm 2** UpdatePrototypes $(x_{new}, P_i, T_i)$

**Input:** $x_{new}$: new coming data instance in cluster $i$,
$P_i$: current prototypes set of cluster $i$,
$T_i$: threshold value for cluster $i$,
**Output:** Updated Prototypes: $P_i^u$
$k \leftarrow |P_i|$; `// Current number of prototypes in` $P_i$
$D_{ij}^{new} =_{j=1,\ldots,k} \|x_{new} - \mu_{ij}\|_2^2$ **if** $D_{ij}^{new} < T_i$ **then**
  `// Update the prototypes` $P_i$
  $\mu_{ij} \leftarrow \frac{1}{n_{ij}+1}(x_{new} + n_{ij} * \mu_{ij})$ $d_{ij} \leftarrow d_{ij} + D_{ij}^{new}$ $n_{ij} \leftarrow n_{ij} + 1$
**else**
  $p_{i,k+1} \leftarrow PrototypeInitial(x_{new})$; `// Algorithm 1`
  $P_i^u = P_i \oplus p_{i,k+1}$
**end**
$T_i^u \leftarrow Update(T_i, P_i^u)$; `// Update the threshold`
**for** $p_{ij}, p_{ik} \in P_i^u$ **do**
  **if** $\|\mu_{ij} - \mu_{ik}\|_2^2 < T_i^u$ **then**
    $\mu_{ij} = \frac{\mu_{ij}+\mu_{ik}}{2}$ $d_{ij} \leftarrow d_{ij} + d_{ik}$ $n_{ij} \leftarrow n_{ij} + n_{ik}$ $P_i^u = Remove(p_{ik}, P_i^u)$
  **end**
**end**
**return** $P_i^u$

### 3.2.2 Labeling Functions Self-Adaption

In this section, we introduce how the built labeling functions adapt to the dynamic datasets. Flame can update the prototypes of the labeling functions and it can also build new labeling functions for those new coming labels.

**Labeling functions update.** Since each labeling function is consisted by a number of prototypes, the labeling functions can be updated by updating its prototypes. We use a threshold to determine whether a new coming data instance $x_{new}$ can be associated to any of the existing prototypes in $P_i$ of cluster $i$. If $x_{new}$ is close to a prototype $p_{ij} \in P_i$, then we update $p_{ij}$. If not, we create a new prototype using Algorithm 1 and add it to $P_i$. Algorithm 2 describes the prototype update process. We first compute a threshold $T_i$ associated with the cluster $i$ using all its existing prototypes, i.e., $T_i = \text{mean}(d_{ij}) + 0.5 * \text{std}(d_{ij})$ for all $p_{ij} \in P_i$. Here, *mean* and *std* are the mean and standard deviation of sum of squared distances in each prototype of cluster $i$. We then compute the closest prototype for the new coming data instance $x_{new}$ by $_{j=1,\ldots,k}D_{ij}^{new}$, where $D_{ij}^{new} = \|x_{new} - \mu_{ij}\|_2^2$, $k$ is the current number of prototypes in $P_i$ (line 2). If $D_{ij}^{new} < T_i$, then the prototype $p_{ij}$ is updated (line 3-6). If not,

then a new prototype is created using $x_{new}$ (line 7-9). The prototype update process may generate a large number of prototypes. Too many prototypes can cause over-fitting, furthermore, storing large number of prototypes consumes too much memory space. To avoid this scenario, we determine whether any two given prototypes in $P_i$ can be merged using the the updated threshold $T_i$ (line 11-16). At last, the updated prototype is returned (line 17).

**New labels detection.** Before building new labeling functions for the new unseen labels, Flame needs to detect the appearance of unseen labels first. Similarly to some existing methods [47], we compute the new unseen labels detection threshold $T^i_{new}$ for each cluster $i$ to reject data instances belong to new unseen labels and assign labels for data instances with existing labels. We refer it as *Nearest Mean Clustering* (NMC).

Due to the dynamic characteristics of the data to be labeled, Flame requires to continuously update prototypes. An optimal threshold value for $T^i_{new}$ should be determined based on the current data patterns. Here, we assume that data of a same label follow a Gaussian distribution. Applying the average inner-cluster distance with a small range of float for each cluster, we obtain the statistic for confidence threshold. A cluster $\mathcal{C}_i$ is consisted by $K$ cliques, the centroids of these cliques are $\mu_{i1}, ..., \mu_{iK}$. For the cluster $\mathcal{C}_i$, we have,

$$dist_i = \frac{1}{\|\mathcal{C}_i\|} \sum_{x \in \mathcal{C}_i} \min_{j=1...K} \|x - \mu_{ij}\|_2^2 \tag{3.5}$$

where $\|\mathcal{C}_i\|$ is the size of the cluster $\mathcal{C}_i$, $\min_{j=1,...,K} \|x - \mu_{ij}\|_2^2$ means the distance of data instance $x$ to it's nearest prototype $p_{ij} \in P_i$. The desired threshold value $T^i_{new}$ for cluster $\mathcal{C}_i$ is calculated by $T^i_{new} = dist_i + \omega * std_i$, where $std_i$ is the standard deviation of $\|x - \mu_{ij}\|_2^2$.

The boundary of a labeling function $lf_i \in LF$ is determined by it prototypes. Each prototype corresponds to a "hypersphere" in the feature space with a centroid and radius. The coverage of a labeling function $lf_i$ is the union of the hyperspheres encompassed by all prototypes in $lf_i$. To assign a label for a new instance $x_{new}$ by the labeling function $lf_i$, we compute the distance set $D^i_{new}$ from $x_{new}$ to the nearest prototype by $\min_{j=1,...,K} \|x - \mu_{ij}\|_2^2$. Finally, we select the minimum distance from set $D^i_{new}$, i.e., $\min D^i_{new}$, If $\min D^i_{new}$ is less than the threshold $T^i_{new}$, the label of $x_{new}$

is from one of the existing labels. Else, $x_{new}$ is regarded as a data instance with an unknown label.

When the data instance $x_{new}$ is from one of the existing labels, we assign the labels for $x_{new}$ as follows. Assume the value of $\min D^i_{new}$'s corresponding prototype is $p_{ij}$. In the prototype $p_{ij}$, $f_{max}$ is the highest frequency value in the frequency vector $\bar{f}$, then $f_{max}$'s corresponding label $y$ will be assigned to data instance $x_{new}$. Each $lf_i \in LF$ maintains an assigned label for the data instance $x_{new}$ and a labeling confidence value (§3.2.3) for its labeling result. The label of the data instance $x_{new}$ is determined by taking the majority vote among all labeling functions.

**Build new labeling functions.** Flame builds new labeling functions for the data instances belong to new unseen labels. When the data instance $x_{new}$ is potentially with a new label, it will be stored into a buffer $B$. Flame periodically checks the buffer $B$ and applies our clustering with minimal impurity (§3.2.1) method on the data instances in buffer $B$ to build the new labeling functions. Therefore, Flame can incrementally incorporate new label information from data instances in $B$. At last, those data instances that belong to the new label are removed from $B$, and the released space is used to collect the subsequent data instances potentially belong to other new labels.

### 3.2.3 Labeling Results Guarantees

**Labeling confidence calculation.** In Flame, each labeling function $lf_i \in LF$ uses a metric ($Conf_i$) to quantify its labeling confidence for a data instance $x$. Flame employs two heuristics, i.e., *association* and *purity* to estimate the $Conf_i(x)$. After each individual labeling function's confidence value $Conf_i(x)(i = 1, .., K)$ acquired, Flame combines these values together to get the entire labeling functions $LF$'s labeling confidence value $Conf(x)$.

The closest prototype from $x$ in labeling function $lf_i$ is $p_{ij}$. The $p_{ij}$ is the $j^{th}$ prototype of $lf_i$, $\ell_{max}$ is the label having highest frequency in $p_{ij}$. The *Association* and *Purity* of the prototype $p_{ij}$ are calculated as follows [46]:

- *Association* is calculated by $R_{ij} - D_{ij}(x)$, where $R_{ij}$ is the radius of $p_{ij}$ and $D_{ij}(x)$ is the distance between $x$ and $p_{ij}$. If $D_{ij}(x)$ is small, it means $x$ is close to the

prototype $p_{ik}$, then $(R_{ij} - D_{ij}(x))$ is large which leads to a high the association and confidence of labeling.

- *Purity* is calculated by $\frac{|L_{ij}(\ell_{max})|}{|L_{ij}|}$, where $|L_{ij}|$ is the sum of all frequencies in $p_{ij}$, and $|L_{ij}(\ell_{max})|$ is the frequency of $\ell_{max}$ in $p_{ij}$. A large $L_{ij}(\ell_{max})$ means the high purity of the prototype $p_{ij}$, which also leads to a high confidence of labeling.

We denote the *Association* and *Purity* of the labeling function $lf_i$ as $A_i$ and $P_i$ respectively. Given the $A_i$ and $P_i$ of the labeling function $lf_i$, its confidence value $Conf_i(x)$ is calculated as below,

$$Conf_i(x) = A_i(x) \times P_i(x), \tag{3.6}$$

**Labeling confidence aggregation.** Flame calculates the confidence value $Conf_i(x)$ for each $lf_i$ in $LF$ for a given data instance $x$. These confidence values are normalized between 0 and 1, and then aggregated together to calculate the overall labeling confidence of all labeling functions as follows,

$$Conf(x) = \max_{\ell \in Y'} \{ \sum_{i=1}^{M} 1(lf_i(x) = \ell) \times Conf_i(x) \} \tag{3.7}$$

where $Conf(x)$ is the aggregated labeling confidence for data instance $x$, $1(lf_i(x) = \ell)$ is an indicator function returns 1 if $lf_i(x) = \ell$ and returns 0 otherwise. After $Conf(x)$ is calculated by Equation 3.7, we have a threshold $\tau$ to decide if $Conf(x)$ is high enough. If it is higher than $\tau$, the label is assigned; Otherwise, the data instance is added into the buffer $B$ for further new unseen labels detection (§3.2.2). The value of $\tau$ is specified by the programmer. We empirically determine $0.6 < \tau < 0.8$ based on the sensitivity study using datasets listed in Table 3.1.

## 3.3 System Design

Flame has four components, *Labeling Functions Generation* (§3.2.1), *Labeling Functions Self-adaption* (§3.2.2), *Labeling Confidence Calculation* and *Labeling Confidence Aggregation* (§3.2.3). Figure 3.3 shows the execution time and energy consumption breakdown for the four components in Flame. Here, "LFG" is short for Labeling Functions Generation, "LFS" is short for Labeling Functions Self-adaptation,

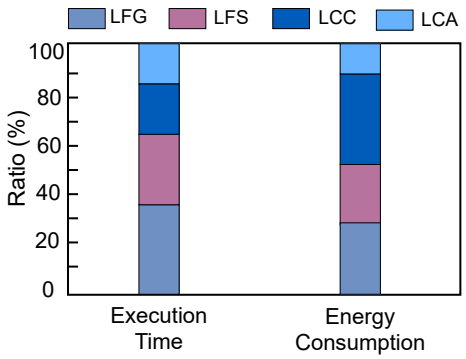


**Figure 3.3:** Breakdown analysis of different components in Flame.

"LCC" is short for Labeling Confidence Calculation, and "LCA" is short for Labeling Confidence Aggregation. The results are received by labeling 3000 data instances in Samsung S9 by Flame. We can see *Labeling Functions Generation* and *Labeling Functions Self-adaption* together consume more than 60% of time, and *Labeling Confidence Calculation* component consumes about 40% of energy. Given a mobile device with three heterogeneous processing units (CPU, GPU and DSP), we map the four components to the three processing units based on the workload characteristics of the four components and hardware. Furthermore, we use a performance model to decide the optimal way to utilize fast shared-memory in GPU when running the workload of *Labeling Functions Generation*. We also use a performance model to coordinate the usage of CPU and DSP while reducing the wake-up rate of CPU for high performance. We discuss our system design in detail as follows.

### 3.3.1 Leverage GPU

Flame uses GPU to run *Labeling Functions Generation* and *Labeling Functions Self-adaption*, because these two components spend more than 60% of time and the clustering with minimal impurity method (§3.2.1) used by these two components is suitable for parallelism. Therefore, we run these two components on mobile GPU to speedup the labeling task. We do not offload the two components to DSP, because they involve heavy computation (such as square root), which cannot be efficiently processed on DSP [48]. To reduce execution of *Labeling Functions Generation*, we make the best use of fast shared-memory on GPU to store data instances and centroids, which brings a challenge. In particular, the fast memory has a rather small

capacity. For example, in our platform (a Samsung S9 mobile phone), shared memory is only 64KB. We store some data instances and centroids in shared memory, such that they do not have to be repeatedly fetched from slow global-memory to build labeling functions. To host as many data instances in fast memory as possible, we apply a sampling method to approximate data instances without impacting labeling accuracy. In particular, given a data instance (an image), we use spatial sampling which selects every $n$-th row for sampling where $n$ is determined based on the image size [49].

There is a non-trivial tradeoff between placing centroids in shared memory and placing data instances in shared memory. To enable high-performance memory accesses to data instances, Flame fetches a batch of data instances ($n_d$) into shared memory and then processes them one by one in shared memory. Leveraging the spatial locality, fetching $n_d$ data instances together causes less global memory accesses. To get the nearest centroid for each data instance in shared memory, Flame must access all centroids. To enable high-performance memory accesses to centroids, Flame also fetches centroids to shared memory batch by batch (the batch size is $n_c$). Placing too many data instances (or too many centroids) in shared memory can cause frequent data movement to fetch centroids (or data instances).

We formulate the above discussion to decide the optimal numbers of data instances ($n_d$) and centroids ($n_c$) to be placed on shared memory as follows. Assume that the total number of data instances to be processed on GPU is $N_d$, the total number of centroids is $N_c$, the execution time of processing $n_d$ data instances and $n_c$ centroids on shared memory is $t$, and the time to transfer $n_d$ data instances and $n_c$ centroids from GPU global memory to shared memory is $t_d$ and $t_c$ respectively. The total execution time $T$ to process $N_d$ data instances and $N_c$ centroids is modeled as follows. We want to minimize $T$ under the constraint of shared memory capacity ($Mem_{shared}$).

$$T = \min_{n_d \geq 0, n_c \geq 0} \lceil \frac{N_d}{n_d} \rceil \cdot t_d + \lceil \frac{N_c}{n_c} \rceil \cdot t_c + \lceil \frac{N_d}{n_d} \rceil \cdot \lceil \frac{N_c}{n_c} \rceil \cdot t, \tag{3.8}$$

$$\text{subject to } \; n_d + n_c \leq \frac{Mem_{shared}}{data_{size}}, n_d \geq 0, n_c \geq 0. \tag{3.9}$$

where $data_{size}$ is the size of a data instance. In Equation 3.8, $\lceil \frac{N_d}{n_d} \rceil \cdot t_d$ denotes the time to transfer $N_d$ data instances from global memory to share memory, $\lceil \frac{N_c}{n_c} \rceil \cdot t_c$ denotes the time to transfer $N_c$ centroids from global memory to share memory, and $\lceil \frac{N_d}{n_d} \rceil \cdot \lceil \frac{N_c}{n_c} \rceil \cdot t$ denotes the execution time to process $N_d$ data instances and $N_c$ centroids on GPU. $t_d$, $t_c$, and $t$ are measured offline. Flame solves the above programming problem using the ALGLIB [50] (a cross-platform numerical analysis and data processing library).

### 3.3.2 Leverage DSP and CPU

Flame runs *Labeling Confidence Calculation* on DSP (not on CPU or GPU), because *Labeling Confidence Calculation* is the most energy-consuming component: It takes 41.6% of energy consumption of the whole auto-labeling workflow. Compared with GPU and CPU, DSP consumes the least power [1]. Furthermore, Flame runs *Labeling Confidence Aggregation* on CPU, not on DSP or GPU. Because this component involves a large amount of division and square root computation, which are not supported effectively on DSP [1]; This component has low thread-level parallelism, making it less efficient to run on GPU either.

The workload execution on CPU and DSP introduces inevitable interaction between CPU and DSP. In particular, the execution of *Labeling Confidence Aggregation* has dependency on *Labeling Confidence Calculation*. Only after *Labeling Confidence Calculation* is done on DSP, *Labeling Confidence Aggregation* can be executed on CPU. We use a common mechanism in mobile phones, the remote procedure call (particularly named *FastRPC* in our evaluation platforms, Samsung S9 and Google Pixel2 mobile phones) for interaction between CPU and DSP.

**Wake up CPU by DSP.** CPU must be woken up by DSP if the execution on DSP takes too long time [51] (e.g., a few seconds) and such wake-up operation on CPU takes 60 mJ, which is very energy-consuming. In Flame, once DSP finishes *Labeling Confidence Calculation* for a batch of data instances, which typically takes 23.6 seconds, the intermediate results generated by DSP must be transferred to CPU cores for *Labeling Confidence Aggregation*. At this moment, DSP must wake up CPU. Compared to the power consumption to wake up CPU, the energy consumption for data transfer between DSP and CPU is relatively small. This is because data transfer

between CPU and DSP consists of simply passing data instances to be labeled, remote invocation parameters, and labeling confidence values calculated by DSP, which is typically 74 KB per transfer and consumes only 3.3 mJ. In general, the rate of waking up CPU is critical to the energy consumption of Flame. We must reduce the wake-up frequency to maintain low-energy consumption.

To minimize the wake-up frequency, we define the CPU wake-up interval and formulate it as follows. The CPU wake-up interval is defined as the time duration from the point where a batch of intermediate results is transferred from DSP to CPU to the point where the next batch is transferred. The CPU wake-up interval heavily depends on memory capacity in DSP and data instance size. Larger memory capacity or smaller data instance size causes longer CPU wake-up interval, and vice versa. Furthermore, DSP processes data instances batch by batch, and processes data instances within the same batch in parallel. Given DSP memory capacity ($Mem_{dsp}$), memory consumed by DSP invocation parameters ($Mem_{para}$), the size of each data instance ($data_{size}$), and the number of threads used by DSP ($n_t$), the number of batches of data instances is $G$, where $G = \lceil \frac{Mem_{dsp} - Mem_{para}}{data_{size} \times n_t} \rceil$. The CPU wake-up interval $\triangle T$ is formulated as follows.

$$\triangle T = \gamma + \sum_{i=0}^{G-1} \{ \max_{j \in [0, n_t - 1]} t_{ij} \} + \rho \tag{3.10}$$

where $\gamma$ is the time to transfer data instances and remote invocation parameters from CPU to DSP, and $\rho$ is the time to transfer calculated labeling confidence values for each data instance from DSP to CPU. The CPU wake-up interval includes the time to process a batch of data instances in DSP, which is formulated as $\max_{j \in [0, n_{th} - 1]} t_{ij}$ in Equation3.10, where $t_{ij}$ is the execution time of $j^{th}$ thread for $i^{th}$ batch of data instances, where $i \in [0, G-1]$ and $j \in [0, n_{th} - 1]$, $n_t$ is the total number of threads in DSP. $\gamma$ and $\rho$ are measured offline. Flame determines the maximum CPU wake-up interval by using ALGLIB on Equation 3.10.

**Data transfer for DSP.** DSP needs to load data to be labeled from CPU main memory to DSP local memory; DSP also needs to transfer labeling confidence values for each data instance between CPU and DSP. To load data from CPU main memory, we use slow CPU cores, because we find that using fast CPU cores often causes a crash on DSP because of a run-out-of-memory error, and using slow CPU cores does

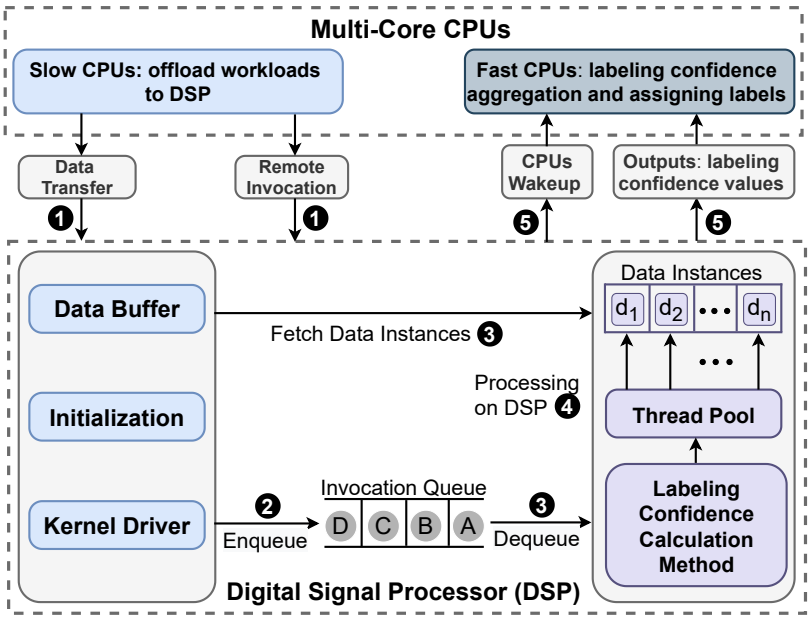


**Figure 3.4:** The interaction between CPU and DSP.

not have this problem. Such an error happens because DSP cannot timely process the data in the local memory before the new data comes in. We use fast CPU cores to transfer data between CPU and DSP.

**Overall workflow.** Figure 3.4 generally depicts the interaction between CPU and DSP. It includes five stages. At Stage one, the slow CPU cores transfer data instances to be labeled and initiates DSP remote invocation. At Stage two, the *FastRPC* kernel driver receives the remote invocations and enqueues them up to wait for the response from DSP. A buffer on DSP is used to store the data instances to be labeled. At Stage three, once DSP is ready for labeling the data instance, it dequeues invocations from the invocation queue and dispatches them for processing. At Stage four, DSP computes the labeling confidence values of data instances in parallel. At last, the labeling confidence values (i.e., the intermediate results) are transferred to fast CPU cores for labeling confidence aggregation and labeling.

### 3.3.3 Implementation

We implement Flame using C++ with Native Development Kit (NDK) on Android 9.0 and Android 8.0. The system is evaluated on two mobile phones (Samsung S9 with Snapdragon 845 SoC and Google Pixel2 with Snapdragon 835 SoC). Our

implementation includes about 8,000 lines of code in total. In our mobile platforms, we have three types of mobile processors, which are GPU, fast CPU and slow CPU. Each mobile platform has mobile GPU, fast CPU, slow CPU, and DSP. To run a workload on a specific type of CPU, we use the thread affinity API. To execute a kernel on GPU, we maintain a CPU thread to execute an OpenCL version of the kernel. To execute a parallel kernel, we examine the availability of CPU cores and GPU at runtime and then obtain the optimal concurrency. The workload running on DSP is implemented by C99. We use FastRPC commonly supported by Samsung S9 and Google Pixel2 for communication between CPU and DSP.

## 3.4 Evaluation

We compare Flame with other baseline methods, in terms of labeling quality and performance. Our evaluation aims to achieve the following four goals.

- Is the labeling quality of Flame better than that of the baseline methods? (§3.4.1)
- Does Flame effectively utilize hardware heterogeneity of mobile processors? (§3.4.2 and §3.4.3)
- Does each component of the Flame effectively boost the overall labeling quality? (§3.4.4)
- Does Flame have imperceivable influences on user experiences on mobile devices? (§3.4.5)

**Experimental Setup.** We use two mobile systems.

- Samsung S9: It uses Qualcomm Snapdragon 845 SoC and Android 9.0 Pie operating system (OS). The 845 SoC includes a 4-core fast CPU, a 4-core slow CPU, an Adreno 630 mobile GPU, and a Hexagon 685 DSP. The fast and slow CPU cores are different in terms of frequency, cache hierarchy, and instruction scheduling. The CPU architecture has 4x Kryo 385 cores (Cortex-A75) at up to 2.8 GHz (max) for performance and 4x Kryo 385 at 1.8 GHz (max) for efficiency.

**Table 3.1:** Characteristics of datasets.

| Datasets | Application | # features | # labels | # instances |
|---|---|---|---|---|
| MNIST | Classification | $28 \times 28$ | 10 | 70,000 |
| EMNIST | Classification | $28 \times 28$ | 62 | 814,255 |
| ImageNet | Classification | $224 \times 224$ | 100 | 60,000 |
| Cifar100 | Classification | $3 \times 28 \times 28$ | 100 | 60,000 |
| UCF50 | Recognition | $320 \times 240$ | 50 | 15,000 |
| UCF101 | Recognition | $320 \times 240$ | 101 | 50,500 |

- Google Pixel2: It uses Qualcomm Snapdragon 835 SoC and Android 8.0 Oreo OS. The 835 SoC includes a 4-core fast CPU, a 4-core slow CPU, an Adreno 540 mobile GPU, and a Hexagon 682 DSP. The CPU architecture has 4x Kryo 280 at 2.45 GHz (max) for performance and 4x Kryo 280 at 1.9 GHz (max) for efficiency.

**Datasets.** Table 3.1 describes the six datasets used to evaluate Flame. Those datasets are commonly used for classification or recognition, which are common applications in mobile devices. To evaluate this ability of Flame, we use the following method. We test Flame on both static and dynamic datasets. (1) Static Datasets. All the labels are known and the data instances are not incrementally generated. (2) Dynamic Datasets. We rearrange instances in each dataset to emulate a dynamic environment where data instances are incrementally generated with previously unseen labels. For MNIST [52], we randomly choose two labels as known, and the rest eight labels as unknown and needed to be detected by Flame. For EMNIST [53], we randomly choose six labels as known, and the rest 56 labels as unknown and needed to be detected. For Cifar100 [54] and miniImageNet [55], we randomly choose ten labels as known, and the rest 90 labels as unknown. UCF50 [56] and UCF101 [57] datasets contain 59, and 101 types of human activities respectively, and they also contain 13,421 short videos created for activity recognition. We use the ffmpeg [58] tool to extract raw images from the above video datasets and feed the unstructured images to Flame sequentially. We select six and ten types of activities as known for UCF50 and UCF101 respectively, and the rest types of activities as unknown and needed to be detected.

For those data instances with known labels, we sample a part of them to form a subset $D_L$. $D_L$ is used to build labeling functions $LF$ for Flame at the beginning

of auto-labeling. $D_L$ takes up to 5% of all the data instances in a dataset. The rest of the dataset ($D_U$) is used to simulate the scenario where new data is incrementally generated for auto-labeling.

**Baselines.** We compare Flame with four baselines: (1) Boosting (AdaBoost), which uses the labeled data to generate one complex decision tree or multiple, simple decision trees to label unlabelled data; (2) Semi-supervised learning [59], which uses both the labeled and unlabeled data to assign labels; (3) Transfer learning, which uses MobileNet_V3 [60] pre-trained on ImageNet for labeling; (4) Snuba [9], a state-of-the-art work for auto-labeling on servers. Snuba cannot run on mobile devices because of the lack of a set of system libraries. So we just report its labeling accuracy.

**Evaluation metrics.** For the dynamic datasets, we use the following metrics to evaluate the labeling quality. (1) $\boldsymbol{Accuracy(\%)} = \frac{N_{new}+N_{exist}}{N}$, where $N_{new}$ is the total number of data instances with unknown labels correctly labeled, $N_{exist}$ is the total number of data instances with known labels correctly labeled, and $N$ is total number of data labeled by the system. Let $FP$ represent the the number of data instances that should be assigned with known labels but is mislabeled with unknown labels (i.e., previously unknown labels); Let $FN$ represent the number of data instances that should be assigned with unknown labels but is mislabeled with known labels; Let $N_l$ represent the number of data instances assigned with unknown labels. We use the following two metrics based on $FP$, $FN$, and $N_l$. (2) $\boldsymbol{M_{new}}$, the percentage of data instances that should be assigned with unknown labels but is mislabeled with known labels, ($M_{new} = \frac{FN\times 100}{N_l}$); and (3) $\boldsymbol{F_{new}}$: the percentage of data instances that should be assigned with known labels but is mislabeled with unknown labels, ($F_{new} = \frac{FP\times 100}{N-N_l}$). Furthermore, we measure the performance of Flame in terms of execution time and energy consumption. We measure the execution time using Flame to label 5000 data instances from each dataset in Table 3.1. We also measure the impact of Flame on user interactions when running Flame on mobile systems.

### 3.4.1 Labeling Quality

In this section, we analysis the labeling quality on both static and dynamic datasets.

**Labeling quality on dynamic datasets.** Table 3.2 shows the results. "ACC"

**Table 3.2:** Comparison of labeling accuracy of different methods.

| Methods | MNIST | | | EMNIST | | | ImageNet | | | Cifar100 | | | UCF50 | | | UCF101 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **Acc** | $M_{new}$ | $F_{new}$ | **Acc** | $M_{new}$ | $F_{new}$ | **Acc** | $M_{new}$ | $F_{new}$ | **Acc** | $M_{new}$ | $F_{new}$ | **Acc** | $M_{new}$ | $F_{new}$ | **Acc** | $M_{new}$ | $F_{new}$ |
| **Boosting** | 62.8 | 78.9 | - | 56.2 | 82.3 | - | 57.2 | 67.8 | - | 64.6 | 72.3 | - | 64.8 | 82.6 | - | 63.4 | 67.3 | - |
| **Transfer** | 76.8 | 62.4 | - | 72.6 | 64.8 | - | 65.7 | 58.3 | - | 70.6 | 69.4 | - | 70.2 | 64.4 | - | 66.7 | 61.9 | - |
| **Semi-s** | 71.5 | 48.3 | 9.8 | 70.7 | 39.7 | 11.3 | 62.4 | 40.4 | 9.4 | 71.7 | 38.4 | 8.9 | 67.2 | 38.3 | 8.8 | 62.4 | 41.7 | 12.9 |
| **Snuba** | 81.2 | 42.5 | 8.3 | 78.2 | 31.2 | 8.5 | 70.0 | 34.7 | 8.7 | 75.3 | 31.8 | 6.4 | 73.3 | 34.8 | 8.2 | 71.2 | 37.1 | 9.8 |
| **Flame_C** | **87.3** | 23.6 | **3.7** | 85.4 | **22.8** | **5.9** | 86.2 | 31.1 | 7.9 | 85.7 | 23.9 | 5.7 | 86.3 | 34.4 | **6.4** | 83.7 | 28.4 | 8.7 |
| **Flme_CG** | 86.2 | **22.7** | 4.5 | **86.1** | 24.7 | 6.2 | 85.8 | **29.9** | **7.6** | **87.5** | 25.0 | **5.2** | 85.7 | **32.3** | 6.6 | 82.4 | 27.7 | 9.7 |
| **Flame_Full** | 86.7 | 22.9 | 5.7 | 85.3 | 23.9 | 6.9 | **87.2** | 30.3 | 8.1 | 85.8 | **23.7** | 6.1 | **87.1** | 33.7 | 6.9 | **84.8** | **26.9** | **8.3** |

is the accuracy of the labeling results; The notation "-" denotes failure of detecting unknown labels. $M_{new}$ is the percentage of data that should be assigned with unknown labels but is mislabeled with known labels; $F_{new}$ is the percentage of data that should be assigned with known labels. For $M_{new}$ and $F_{new}$, a lower value indicates a better result. In general, Flame performs best. We conclude the following. (a) Flame outperforms the boosting and semi-supervised methods. The average labeling accuracy of Flame is higher than that of the two methods by 25.2% and 18.5% respectively. The reason is that Flame has the ability to immediately detect new coming unknown labels. This advantage is obvious when the number of unknown labels in the dataset is large (e.g., ImageNet, Cifar100, UCF101); (b) Flame outperforms the transfer learning method up to by 16.1%. This is because the pre-trained model is directly used for labeling without learning a representation of the data from scratch; (c) Flame outperforms Snuba by 11.8%. Snuba's labeling quality is based on trained labeling functions and it fails to assign unknown labels to data instances, because Snuba's labeling functions can not adapt to dynamic datasets; (d) The labeling quality of different versions of Flame (i.e., CPU Only, CPU and GPU, and the full featured version which uses CPU, GPU and DSP) does not vary significantly. The slight variance in the labeling quality comes from randomness in execution order of parallel threads in the heterogeneous computing environment.

**End-to-end impact**. We evaluate the effect of using the auto-labeled data to train DNN models. In our evaluation, we use a DNN model with three fully-connected layers and each layer contains 128 neurons, which is commonly deployed in mobile phones [61]. We stop the training process until the DNN model's loss value converges. Table 3.3 shows the accuracy lift after using auto-labeled data from Flame. The column "Highest Accuracy of Baseline" means the highest accuracy achieved by

**Table 3.3:** Accuracy comparison between baselines and Flame.

| Datasets | Highest Accuracy of Baseline (%) | Flame (%) | Lift |
|---|---|---|---|
| MNIST | 81.1 | 88.2 | +7.1 |
| EMNIST | 76.6 | 84.1 | +7.7 |
| ImageNet | 75.2 | 84.6 | +9.4 |
| Cifar100 | 72.3 | 79.6 | +6.3 |
| UCF50 | 78.3 | 85.2 | +6.9 |
| UCF101 | 76.5 | 83.7 | +7.2 |

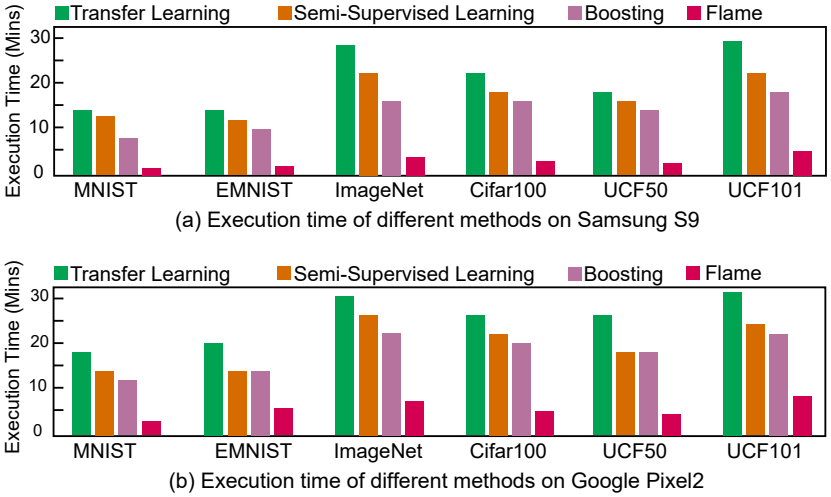


**Figure 3.5:** Comparison between different labeling methods in execution time.

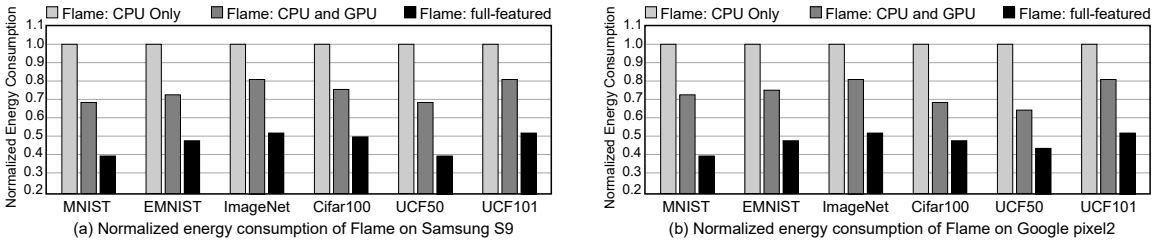


**Figure 3.6:** Comparison of the energy consumption.

the baselines. The accuracy lift over the six datasets shows that labeling results of Flame improves the model accuracy. In general, the model accuracy is improved by up to 9.4% over six datasets.

### 3.4.2 Analysis on Execution Time

We compare the execution time of different labeling methods. We also show how our system design (§3.3) can reduce the execution time of Flame.

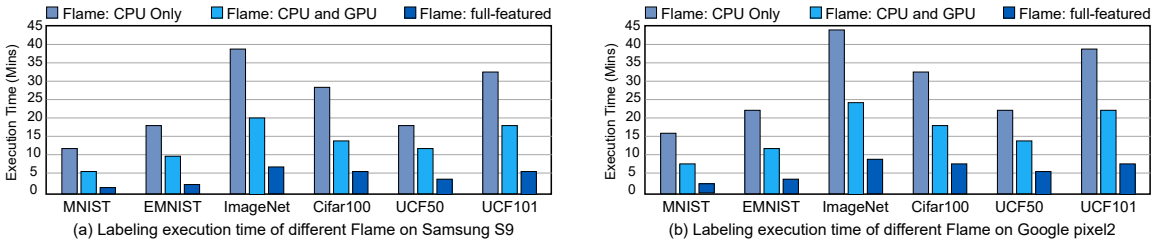


**Figure 3.7:** The execution time of the three versions of Flame.

**Comparison with different versions of Flame.** Figure 3.7 presents execution time of labeling 5,000 data instances on Samsung S9 and Google Pixel2. We use three execution strategies to evaluate the effectiveness of Flame: (1) using CPU Only; (2) using CPU and GPU; and (3) using CPU, GPU and DSP (i.e., the full-featured Flame). Figure 3.7 shows that compared with using CPU Only, using CPU and GPU leads to an average of 2.1× speedup, because of using GPU. Using the full-featured Flame, there is average 6.8× performance improvement, because of using GPU and DSP. Flame fully taps the capability of hardware heterogeneity in mobile processors. Such a large reduction in execution time is not paid by using larger energy consumption, discussed as follows.

**Comparison with the baselines.** Figure 3.5 presents the execution time of labeling 5,000 data instances on Samsung S9 and Google Pixel2 using different methods. Compared with the baselines, Flame's execution time is the shortest. Figure 3.5 shows that Flame reduces the execution time by average 5.6×, 4.2×, and 3.8× respectively, compared with the transfer learning, semi-supervised learning, and the boosting methods. Such a reduction in execution time is because of two reasons: (1) Flame uses the lightweight clustering with minimal impurity method (§3.2.1) to build the labeling functions; and (2) Flame uses Algorithm 2 to merge the close prototypes to reduce the number of prototypes contained in each labeling function, which significantly reduces computation overhead in Flame.

### 3.4.3 Analysis on Energy Consumption

We analysis the energy consumption of different labeling methods over the six datasets. We also analyze how full-featured Flame saves energy, compared with the other two versions of Flame.

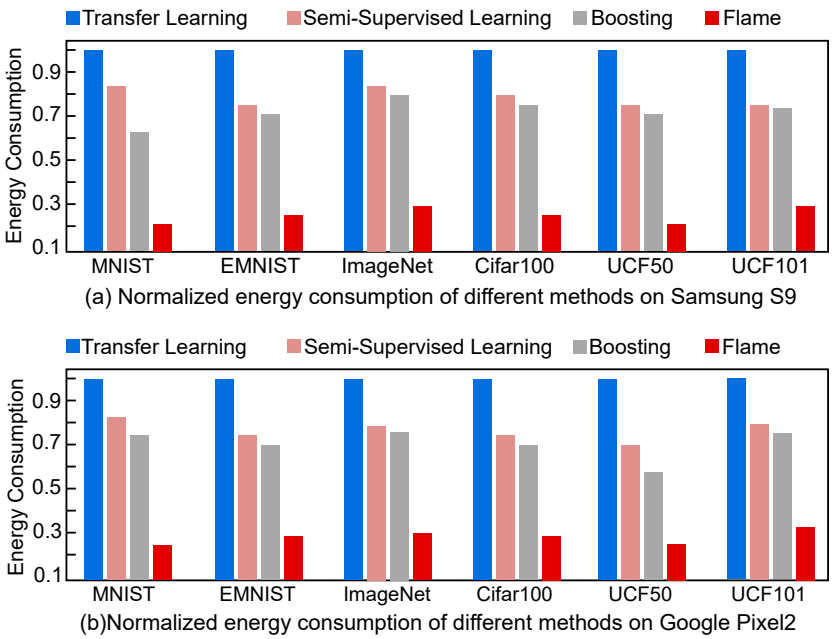


**Figure 3.8:** Energy consumption comparison of different labeling methods.

**Table 3.4:** Ablation study results of different components in Flame.

| Datasets | Labeling Functions Generation | | | Labeling Functions Self-adaption | | | Labeling Results Guarantees | | |
|---|---|---|---|---|---|---|---|---|---|
| | Accuracy | $M_{new}$ | $F_{new}$ | Accuracy | $M_{new}$ | $F_{new}$ | Accuracy | $M_{new}$ | $F_{new}$ |
| MNIST | +14.4 | -5.1 | -8.3 | +7.8 | -7.2 | -5.1 | +6.5 | -4.2 | -3.5 |
| EMNIST | +18.5 | -8.8 | -12.2 | +12.5 | -18.3 | -17.1 | +9.1 | -7.7 | -6.4 |
| ImageNet | +21.3 | -7.8 | -13.7 | +14.2 | -25.4 | -14.4 | +9.1 | -8.2 | -8.4 |
| Cifar100 | +24.1 | -10.4 | -13.1 | +14.6 | -27.3 | -20.6 | +12.9 | -8.8 | -7.7 |
| UCF50 | +17.7 | -9.2 | -11.0 | +8.7 | -13.5 | -11.5 | +11.1 | -7.2 | -6.1 |
| UCF101 | +19.5 | -7.4 | -12.8 | +9.7 | -23.3 | -18.6 | +12.7 | -7.4 | -9.8 |

**Comparison with different versions of Flame.** Figure 3.6 shows the energy consumption of the three strategies (CPU Only, CPU and GPU, and the full-featured Flame) on Samsung S9 and Google Pixel2 . The energy consumption in Figure 3.6 is normalized by that of running Flame on Google Pixel2 with only CPU. Figure 3.6 shows that the average energy consumption of using the strategies of "CPU and GPU" and full-featured Flame is 73.4%, and 46.2% of that of Flame using only CPU, respectively. Full-featured Flame uses the least energy, which shows the advantage of using DSP.

**Comparison with the baselines.** Figure 3.8 shows energy consumption of the semi-supervised learning, transfer learning, boosting, and Flame. The energy results

are normalized by energy consumption of using the transfer learning method on each dataset. In Figure 3.8, energy results are normalized by the energy consumption of using the transfer learning. In general, Flame consumes the least energy in all datasets. This is largely because we offload the most energy-consuming component of Flame (*Labeling Confidence Calculation*) to DSP (§3.3.2).

### 3.4.4 Micro-Benchmarking Results

We evaluate the components of Flame (i.e., *Labeling Functions Generation*, *Labeling Functions Self-adaption*, and *Labeling Results Guarantees*) and show how each component can affect the auto-labeling quality. Table 3.4 shows the results. We measure how the evaluation metrics change when involving different components in Flame. $M_{new}$ is the percentage of data that should be assigned with unknown labels but is mislabeled with known labels; $F_{new}$ is the percentage of data that should be assigned with known labels but mislabeled with unknown labels.

**Labeling functions generation.** We quantify the contribution of *Labeling Functions Generation* (§3.2.1) to the labeling quality. Table 3.4 shows that (1) compared with the other two components, *Labeling Functions Generation* contributes more to the labeling accuracy, and (2) including *Labeling Functions Generation* improves the accuracy by up to 24.1% (Cifar100). We conclude *Labeling Functions Generation* component is useful for improving the labeling accuracy, especially for those datasets with a large numbers of unknown labels (e.g., Cifar100 and ImageNet). Additionally, $M_{new}$ and $F_{new}$ are decreased by up to 10.4% and 13.1% respectively after involving *Labeling Functions Generation*. Therefore, this component also boosts the capability of recognizing unknown labels and known labels.

**Labeling functions self-adaption.** We quantify the contribution of *Labeling Functions Self-adaption* component (§3.2.2) to the labeling quality. Table 3.4 shows that (1) *Labeling Functions Self-adaption* leads to more reduction in $M_{new}$ and $F_{new}$ than the other two components, indicating that it is more efficient to recognize unknown labels and known labels than the other two components. (2) *Labeling Functions Self-adaption* is more helpful to those datasets with a larger number of unknown labels. For example, Cifar100 has 100 labels and MNIST has 10 labels. The reduction of $M_{new}$ and $F_{new}$ on Cifar100 dataset is 27.3% and 20.6% respectively,

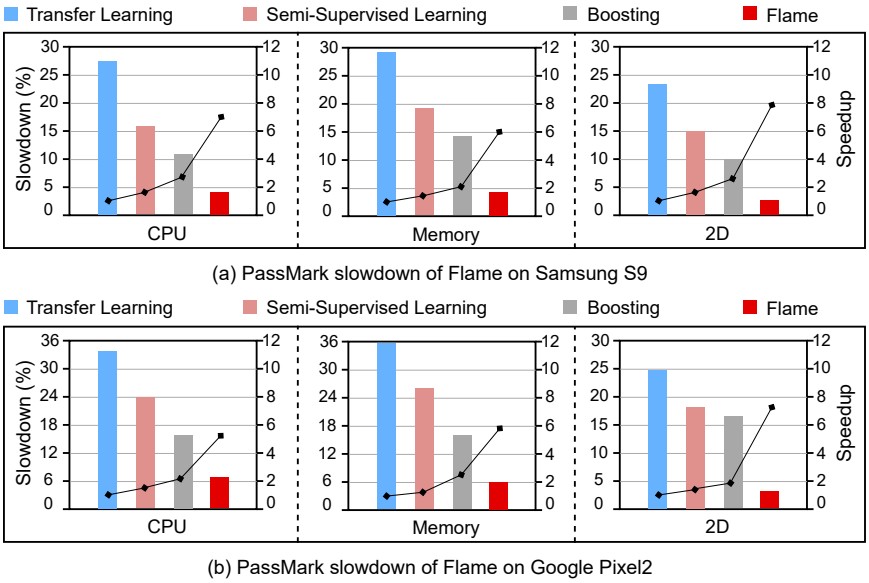


**Figure 3.9:** PassMark slowdown with auto-labeling running in background.

while the reduction on Labelme dataset is only 7.2% and 5.1% respectively.

**Labeling Results Guarantees.** We quantify the contribution of *Labeling Confidence Aggregation* (§3.2.3) to the labeling quality. Table 3.4 shows that (1) *Labeling Confidence Aggregation* improves the accuracy by up to 12.9%, and (2) $M_{new}$ and $F_{new}$ are decreased by up to 8.8% and 9.8% respectively after involving *Labeling Confidence Aggregation*.

### 3.4.5 Evaluation on User Experience

We run Flame in background as a service in mobile phones, in order to avoid the impact of using Flame on the user applications running in foreground. In this section, we evaluate the impact of using Flame on the user experience. Android always sets higher priority to the foreground applications compared to the background applications to provide prompt response to user input, more resources are allocated to the foreground applications. Since labeling is an intensive task, running it in background can reduce its impact on user experience of other applications on the devices. Therefore, We run Flame in background.

**Impact on other applications.** We evaluate how using different labeling methods impacts the performance of another application running in foreground. We use a benchmark PassMark [62] as the user application. We use PassMark, because it

involves CPU tests, memory tests, and graphics tests, representing workloads with various characterization. Figure 3.9 shows the slowdown of PassMark while running various labeling methods in background. We compare Flame with other labeling methods in terms of PassMark slowdown. In general, Flame has the least impact on PassMark. (a) For the CPU tests, the full-featured Flame leads to up to 4.6% and 6.1% slowdown in PassMark on Samsung S9 and Google Pixel2 respectively. The CPU slowdown of the full-featured Flame is up to 7.1× better than the baselines. (b) For the memory tests, the full-featured Flame leads to up to 4.7% and 5.9% slowdown in PassMark on Samsung S9 and Google Pixel2 respectively. Compared to the baselines, the full-featured Flame achieves up to 5.7× and 5.9× slowdown improvement on Samsung S9 and Google Pixel2 respectively. (c) For 2D graphic experience, the full-featured Flame leads to up to 3.1% and 4.3% slowdown in PassMark on Samsung S9 and Google Pixel2 respectively. Compared to the baselines, the slowdown improvement for 2D graphic experience is up to 7.9× and 7.4× on Samsung S9 and Google Pixel2 respectively.

**Impact on the interaction between the user and mobile devices.** We aim to find out whether running Flame affects a user's interactive experience with the mobile device. We perform our tests using an application that models the user interaction with a device by taking a user's input from the touch screen and rendering a response on the screen. By analyzing the rendering latency in response to the user input, we can quantitatively understand the interactivity. We use Android dumpsys tool to measure the latency. We use two metrics to quantify the latency: (1) response time, i.e., the time for the mobile device to process the user input; and (2) frame latency, i.e., the time to render a new frame based on the user input. Figure 3.10 lists the 95th percentile response time and Figure 3.11 lists the frame latency distribution under four different configurations. These configurations are (1) without running Flame (baseline), (2) running Flame on CPU (Flame: CPU Only), (3) running Flame on CPU and GPU (Flame: CPU and GPU), and (4) running Flame on CPU, GPU and DSP (Flame: full-featured).

Figure 3.10 shows that the response time increases by only 0.03 ms and 0.04 ms when running full-featured Flame on Samsung S9 and Google Pixel2. The response time in the figure shows how fast the application reacts to user input events. Meanwhile, as Figure 3.11 shows, the median frame latency only increases by 0.75

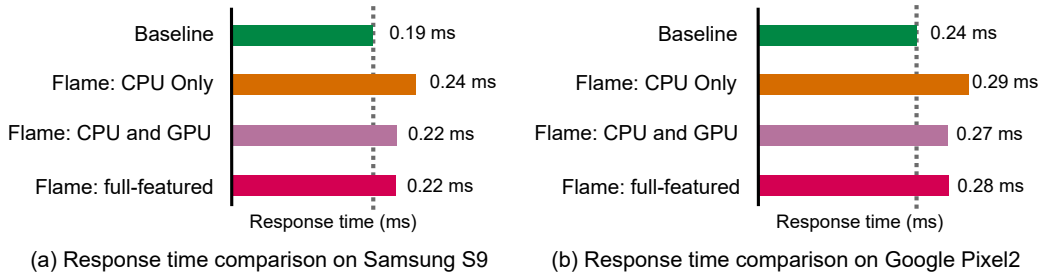


**Figure 3.10:** Impacts of using Flame on user experience.

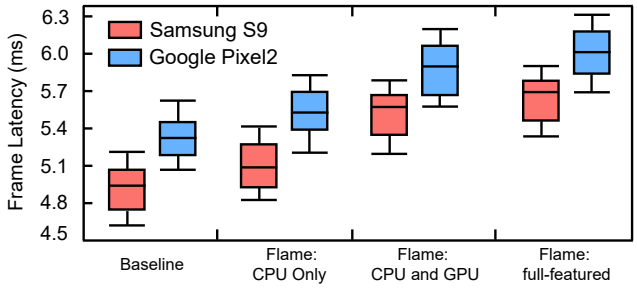


**Figure 3.11:** The frame latency distribution of using different versions of Flame.

ms and 0.64 ms when running full-featured Flame on Samsung S9 and Google Pixel2 respectively. Such increase is very small, compared with the minimum threshold of user-perceivable latency, 100 ms [11]. Baseline is the frame latency without running Flame. Figure 3.11 also shows that the additional frame latency caused by Flame: the full-featured Flame causes negligible latency, compared with using CPU and GPU to run Flame. This means that involving DSP into Flame does not affect the user's graphic experience. For the above testing results, we conclude that the impact of the labeling tasks is not perceivable by users, because the minimum threshold of perceivable latency is 100 ms [11].

## 3.5 Summary

Auto-labeling on mobile devices is critical to enable ML training on mobile devices. However, it is challenging to enable auto-labeling on mobile devices, because of unique data characteristics on mobile devices and heterogeneity of mobile processors. In this paper, we introduce the first auto-labeling system for mobile devices, named Flame, to address the above problem. Flame includes auto-labeling algorithms to detect unknown labels from dynamic data.

# Chapter 4

# Fauce: Fast and Accurate Deep Ensembles with Uncertainty for Cardinality Estimation

## 4.1 Overview of Fauce

Figure 4.1 shows the architecture of Fauce at a high level. The Query Featurization (§4.3) transforms the queries into vectors, it includes Tables Encoding (§4.3.1), Joins Encoding(§4.3.1), Columns Encoding (§4.3.2), and basic statistical information (§4.3.3). The generated training dataset (§4.4.2) is used to train the appropriately designed regression model (§4.5). At last, the trained model is used to estimate the query cardinalities (§4.5.2).

Fauce consists of two stages. First, Fauce transforms input queries into feature vectors through a new query featurization method (§4.3), including tables encoding and joins encoding). Tables encoding (§4.3.1) is based on a graph embedding method that can capture semantic information of the database tables and achieve more accurate encoding results than widely used one-hot encoding and binary encoding methods. Joins encoding (§4.3.1) is based on our proposed *joins2vec* algorithm to featurize joins into vectors. Without any assumption on the independence of columns, our column encoding (§4.3.2) can capture real dependency information among the columns. Besides the encoding information, Fauce also collects statistics of the database tables (e.g., row counts and domain bounds) to represent the point predicate and/or range

predicate of a query (§4.3.3).

Second, we train the model $\mathcal{M}$ based on the generated training dataset (§4.4.2). Once the training is finished, the model is ready to estimate the cardinalities for a given query. For each input query, we use a query featurization method to transform the query into a feature vector. This vector is plugged into the model $\mathcal{M}$, and the output of $\mathcal{M}$ is the estimated cardinality together with the corresponding uncertainty.

## 4.2 Problem description

In this section, we introduce some notations and describe why the cardinality estimation can be solved as a regression problem.

### 4.2.1 Notations

Consider a database $\mathcal{D}$ contains $m$ tables, $\mathcal{D} = \{T_i\}_{i=1}^{m}$. Each table $T_i$ has a number of numeric columns, represented as $T_i = \{Col_i^1, ..., Col_i^{c_k}\}$, where $c_k$ is the total number of columns in Table $T_i$. The total number of columns in $\mathcal{D}$ is denoted as $C$, where $C = \sum_{k=1}^{m} c_k$.

We define the actual cardinality of a query $q$ as the number of rows in joint tables that satisfy all predicates in $q$, and denote it as $Act(q)$. Similarly, we use $Card(q)$ to represent the estimated cardinality for the query $q$. Each query $q$ can be represented as a collection of four sets: $\langle Tables \rangle$, $\langle Joins \rangle$, $\langle Columns \rangle$, $\langle Values \rangle$, and each set is defined as below.

- $\langle Tables \rangle$: the set of the tables in $q$'s FROM clause.
- $\langle Joins \rangle$: the set of the join relations in $q$'s WHERE clause.
- $\langle Columns \rangle$: the set of the columns involved in $q$'s WHERE clause.
- $\langle Values \rangle$: the set of the predicates values in $q$'s WHERE clause.

These four sets together depict the features of a query.

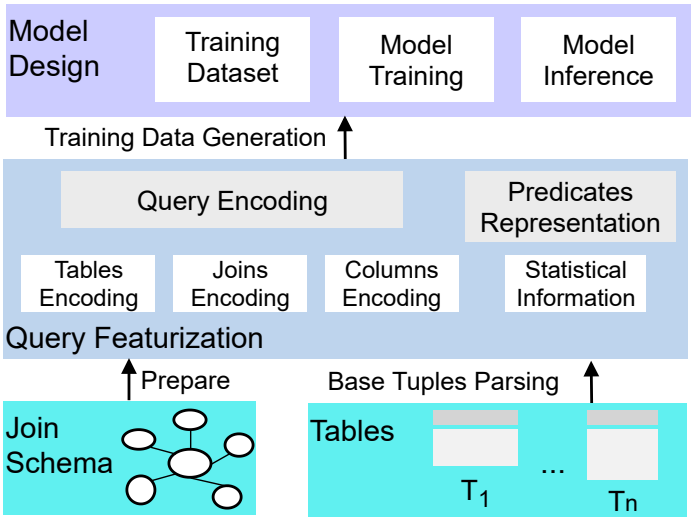


**Figure 4.1:** Overview of Fauce.

### 4.2.2 Formulation as a Regression Problem

As the cardinality of a query is a real-valued number, we develop a regression model $\mathcal{M}$, such that for any range query $q$ on joint tables, the estimated cardinality $Card(q)$ produced by $\mathcal{M}$ matches or closes to the actual cardinality $Act(q)$.

The input of the model $\mathcal{M}$ must be a real-valued vector. Therefore, we must transform the query $q$ into a real-valued vector which represents the features of $q$. This transformation is called *query featurization*. For a query $q = \langle Tables \rangle$, $\langle Joins \rangle$, $\langle Columns \rangle$, $\langle Values \rangle$, we transform $q$ into a query feature vector $\vec{f} = \langle f_T, f_J, f_C, f_V \rangle$, where $f_T$, $f_J$, $f_C$, and $f_V$ are the features extracted from $\langle Tables \rangle$, $\langle Joins \rangle$, $\langle Columns \rangle$, and $\langle Values \rangle$ respectively. The vector $\vec{f}$ serves as the input to the regression model $\mathcal{M}$. The actual cardinality, $Act(q)$, serves as the labels, which guides the model training. Given a training set of labeled queries $\mathcal{S}$, the model $\mathcal{M}$ trained on $\mathcal{S}$ is expected to produce accurate cardinalities for unseen queries.

## 4.3 Query Featurization

Before using the model $\mathcal{M}$ to estimate the cardinality, we must convert input queries into vectors. A query $q$ can be represented as: $\langle Tables \rangle$, $\langle Joins \rangle$, $\langle Columns \rangle$, and $\langle Values \rangle$. Each of them is represented by a vector. These four vectors combined together is the outcome of the *query featurization* for $q$. The result is directly plugged

into the model for both training and inference. Section 4.3.1 introduces how to encode $\langle Tables \rangle$ and $\langle Joins \rangle$ into vectors; Section 4.3.2 introduces the method to encode $\langle Columns \rangle$; and Section 4.3.3 introduces how to represent $\langle Values \rangle$ of a query.

### 4.3.1 Tables and Joins Encoding

**Tables encoding.** Instead of using one-hot and binary encoding methods, we use a graph embedding method [63] to encode the database tables. The *join schema* of a database is considered as an undirected graph $G$, where vertices are tables and each edge connects two joinable tables. We use $G$ as the input for the graph emebedding method, and the output is a group of vectors. Each vector is the encoding result for a corresponding table. In a database $\mathcal{D} = \{T_i\}_{i=1}^{m}$, if a table is not involved in a query, we use a vector with all zeros to represent this table. Similar to the binary encoding, our tables encoding method represents each table as a $\lceil \log(m+1) \rceil$ dimensional vector, where $m$ is the number of tables in a database. Finally, the $\langle Tables \rangle$ of a query $q$ is represented as a vector $f_T$ with length of $m\lceil \log(m+1) \rceil$.

**Joins encoding.** Using the existing coarse-grained joins encoding methods [16] for *query featurization* always causes large errors in cardinality estimation. We propose a new fine grained algorithm called *Joins2Vec* for the joins encoding.

**Algorithm 3** GetJoinGraphs $(G, t, d)$

**Input:** $JS = (V, E)$: The join schema of a database
$t$: Table which is the root of a join relationship
$d$: Neighbours considered for extracting join graph
**Output:** $jg_t^{(d)}$: rooted join graph of degree $d$ around table $t$
$jg_t^{(d)} = \{\}$ **if** $d = 0$ **then**
  $jg_t^{(d)} := t$
**else**
  $\mathcal{N}_t := Neighbours(G, t)$; `// Breath First Search`
  $M_t^d := \{\text{GetJoinGraphs}(G, t', d-1) | t' \in \mathcal{N}_t\}$
  $jg_t^{(d)} := jg_t^{(d)} \cup \text{GetJoinGraphs}(G, t, d-1) \oplus M_t^d$
**end**
**return** $jg_t^{(d)}$

The $\langle Joins \rangle$ of a query $q$ is based on the *join graphs* derived from the join schema $JS$. The Algorithm *Joins2Vec* consists of two main components; the first component discovers all the possible *join graphs* based on the join schema $JS$, and the second

component gets the encodings for all the *join graphs*. The goal of Algorithm *Joins2Vec* is to learn a $\lambda$ dimensional encoding for each *join graph*. We first search all the join graphs, $JGs$ (extensive details are depicted in Algorithm 3). Then the encodings for the join graphs in $JGs$ are initialized as a matrix: $\Theta \in \mathbf{R}^{|JGs| \times \lambda}$ where $|JGs|$ is the number of possible join graphs extracted from $JS$. After that, we learn the encoding result $\Theta$. These steps are explained in detail in the following two paragraphs.

**(1) Get all the join graphs.** First, we introduce how to use each table $t$ in the database $\mathcal{D}$ as a root to build the join graphs. The join graph $jg_t^{(d)}$ rooted at the table $t$ with different numbers of joinable tables $d$ in a given join schema $JS$ is extracted. The join graphs discovering process is separately explained in Algorithm 3. The Algorithm 3 takes the join schema $JS$, table $t$, and degree of the joins $d$ as inputs and returns the intended join graph $jg_t^{(d)}$. When $d = 0$, no join graphs need to be extracted and the table $t$ is returned. For the case when $d > 0$, we get all the (breadth-first) neighbours of $t$ in $\mathcal{N}_t$, and the neighbours of $t$ are those tables that can be joined with the table $t$. Then for each joinable table, $t^{'}$, we get its $(d-1)$-degree join graphs and save them in $M_t^{(d)}$, where $M_t^{(d)}$ is a list to store the rooted d-degree join graphs around table $t$. Finally, we get the $(d-1)$-degree join graph around the table $t$ and concatenate these join graphs with $M_t^{(d)}$ to obtain the intended join graphs $jg_t^{(d)}$.

**(2) Get the context for each join graph.** Then, we introduce how to get the context for each join graph based on the results of Algorithm 3. Once the join graphs $jg_t^{(d)}$ of table $t$ is extracted, we learn the encoding of a target join graph using its surrounding context in a given join schema $JS$. We define the context of a $d$-degree join graph $jg_t^{(d)}$ of the table $t$ as the set of join graphs of $(d-1)$, $d$ and $(d+1)$-degree rooted at each of the neighbours of $t$. Note that we consider join graphs of $(d-1)$, $d$ and $(d+1)$-degree to be in the context of a join graph of $d$-degree, because a $d$-degree join graph is likely to be rather similar to the join graphs of degrees that are closer to $d$ (e.g., $d-1$ and $d+1$) and not just the $d$-degree join graphs only.

**(3) Optimize the encodings for the join graphs.** The encoding of a target join graph, $jg_t^{(d)}$, with the context $context_t^{(d)}$ is learnt using the process in algorithm *Joins2vec*. Given the current representation of the target join graph $\Theta(jg_t^{(d)})$, we want to maximize the probability of every join graph in its context $jg_{cont}$. Here, we learn such posterior distribution using logistic regression classifier. Finally, the encodings of all the join graphs are optimized by gradient descent.

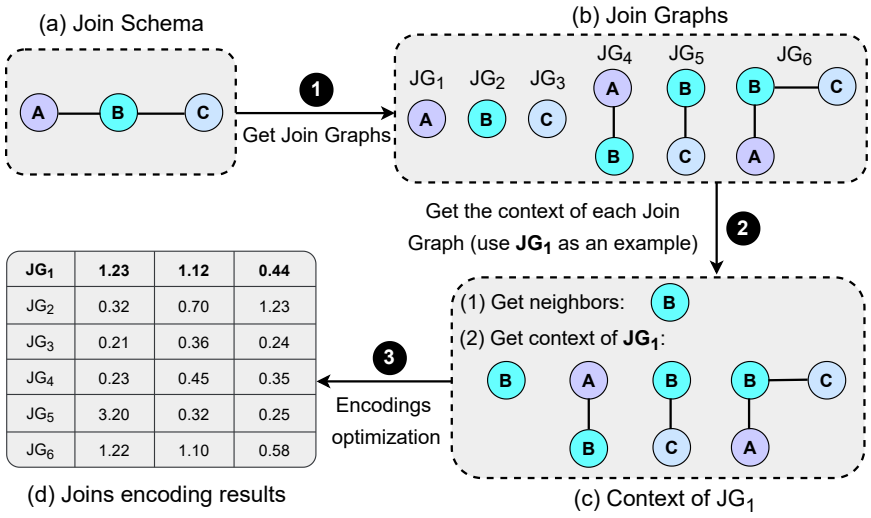


**Figure 4.2:** An example of *Joins2Vec*.

Using Algorithms: *Joins2Vec* and *GetJoinGraphs*, we get the encoding result for $\langle Joins \rangle$ of a query. Assume there are $n$ possible join graphs in a database and the encoding size $\lambda$ is equal to $n$, the representation of $\langle Joins \rangle$ of the query $q$ is a $n$ dimensional vector. Figure 4.2 shows an example of applying *Joins2Vec* on a join schema with three tables, `A`, `B`, and `C`. In this figure, (a) A join schema with three tables. (b) Get all the possible join graphs based on (a) using Algorithm 3. (c) Get the context of each join graph. (d) Encode the join graphs into vectors. All the join graphs derived from this join schema are encoded into vectors (see (d) in Figure 4.2).

### 4.3.2 Columns Encoding

The correlations of table columns can be utilized as useful information to facilitate the columns encoding. We propose a method called *Columns2Vec*, which encodes the columns by using real correlations among the columns. This method includes three steps.

**(1) Build local columns-dependency graphs.** We calculate the Randomized Dependence Coefficient [64] (RDC) values for each pair of columns in each table $T_i$ from the database $\mathcal{D}$. If the RDC value for two columns exceeds a threshold $\tau$, then those two columns are dependent with each other; otherwise, they are independent. Using a small value of $\tau$ overestimates the columns-dependency, while using a large value of $\tau$ underestimates the columns-dependency. Here, we set $\tau$ as 0.4. Based on

the RDC values of each pair of columns, we can build a local columns-dependency graph $g_i$ for each table $T_i$. The graph $g_i$ is a DAG. Once there exists a connection (i.e., an edge) between two columns (i.e., vertices), the graph shows those two columns are correlated. We get dependency information between any pair of columns in $g_i$ by using depth first search to find whether a path exists between their corresponding vertices.

**(2) Build a global columns-dependency graph for the database.** This graph is represented as $G$. It is built based on the local column-dependency graphs. Assume that there are two tables $T_i$ and $T_j$ and their local columns-dependency graphs are $g_i$ and $g_j$ respectively. We merge $g_i$ and $g_j$ if $T_i$ and $T_j$ are joinable. Thus, we can build the global columns-dependency graph $G$ by checking whether the pair of tables $T_i$, $T_j$ in a database are joinable or not.

**(3) Use graph embedding for encoding.** We use a graph embedding [15] to encode each vertex in $G$ into a vector. The results of *Columns2Vec* are used to represent $\langle Columns \rangle$ of a query $q$ as a vector $f_C$ with the length $C$, where $C$ is the total number of different columns in the database. Figure 4.3 shows an example of applying *Columns2Vec* to three tables, `Title`, `Company`, and `Cast`. In this figure, (a) Three tables, and their columns to be encoded. (b) Local columns-dependency graph for each table, vertices are columns, an edge represents the correlations between two columns. (c) Global columns-dependency graph. (d) Use the graph from (c) as the input for the graph embedding method. Finally, each column in the database is represented as a vector. Multiple columns from these three tables are encoded into vectors (see (d) in Figure 4.3).

### 4.3.3 Range Representation

In this section, we discuss how to represent $\langle Values \rangle$ of a query. In a database $\mathcal{D} = \{T_i\}_{i=1}^{m}$, any conjunctive query $q$ on numeric columns of the database $\mathcal{D}$ can be represented as a subset of $(lb_1^1 \leq Col_1^1 \leq ub_1^1) \wedge ... \wedge (lb_m^{c_m} \leq Col_m^{c_m} \leq ub_m^{c_m})$, where $Col_i^j$ is the $j^{th}$ column of the table $T_i$, $lb_i^j$ and $ub_i^j$ are the lower bound and upper bound on values in the column $Col_i^j$ respectively, and $\{c_i\}_{i=1}^{m}$ is the number of columns in the tables $\{T_i\}_{i=1}^{m}$. Let the domain of the $j^{th}$ column in table $T_i$ be $dom(Col_i^j) = [min_i^j, max_i^j]$. If a query does not contain predicate on column $Col_i^j$,

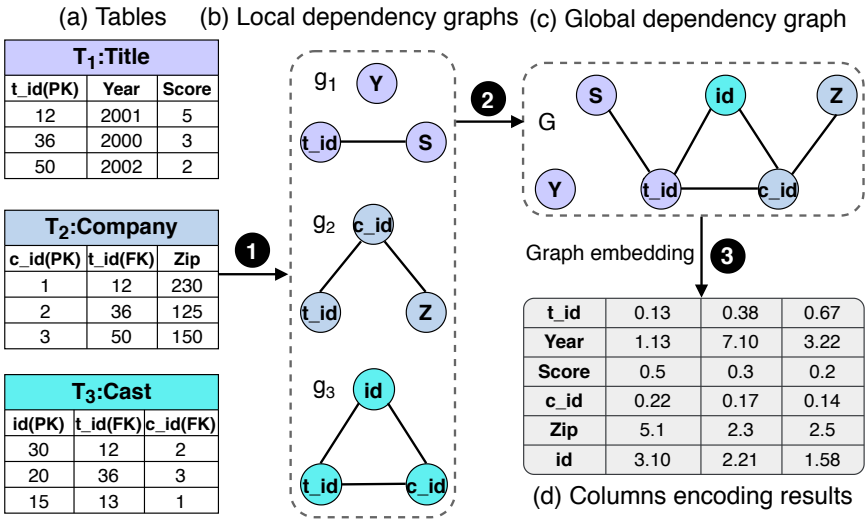


**Figure 4.3:** End-to-end example of *Columns2Vec*.

then we have $lb_i^j = min_i^j$ and $ub_i^j = max_i^j$. Then, the predicate on the column $Col_i^j$ becomes $min_i^j \leq Col_i^j \leq max_i^j$. It means that the predicate on $Col_i^j$ does not filter out any row. For instance, assume that there are two columns $Col_i^j$ and $Col_i^k$ from the table $T_i$, each of which is in the domain [0,100]. Then, the predicate $10 \leq Col_i^j \leq 20$ would have the following representation: $(10 \leq Col_i^j \leq 20) \wedge (0 \leq Col_i^k \leq 100)$. The above definition includes one-sided range predicates and point predicates, i.e., $Col_i^j = x$ can be specified as $lb_i^j = x$ and $ub_i^j = x$.

Finally, we use a vector $f_V$ with $2C$ dimensions to represent $\langle Values \rangle$ of a query, it is represented as: $\langle lb_1^1, ub_1^1, ..., lb_m^{c_m}, ub_m^{c_m} \rangle$. This vector is used as a part of input features for the model $\mathcal{M}$. To facilitate learning, all the vectors constructed by the *query featurization* have the same dimension and the same format as depicted in Table 4.1, where $m$ is the number of tables in the database, $n$ is the number of possible join relationships among the database tables, and $C$ is the total number of different columns in the database. Therefore, the feature vector for the query $q$ after the *query featurization* has a length of $L = m\lceil \log(m+1) \rceil + n + 3C$.

## 4.4 Choice of regression methods

We use an ensembles of *deep neural networks* (DNNs), or *deep ensembles* for short, to estimate the cardinalities. We choose DNN, because the distribution of queries is very complex and DNNs are powerful models that have achieved impressive

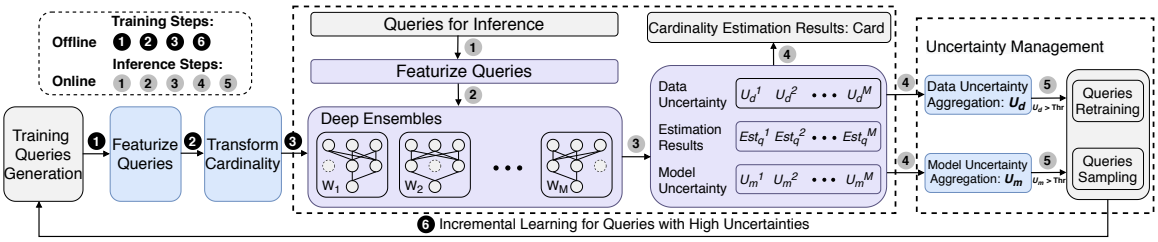


**Figure 4.4:** Training and inference process of Fauce.

**Table 4.1:** Query Features Segmentation

| Type | Table | Join | Column | Predicate |
|---|---|---|---|---|
| Segment | $\langle Tables \rangle$ | $\langle Joins \rangle$ | $\langle Columns \rangle$ | $\langle Values \rangle$ |
| Method | Embedding | Joins2Vec | Columns2Vec | Range |
| Seg. Size | $m\lceil \log(m+1) \rceil$ | $n$ | $C$ | $2 \times C$ |

accuracy on many tasks. Furthermore, previous work [65, 66] has shown the advantage of using the ensemble technique to boost the cardinality estimation.

**Deep ensembles.** *Deep ensembles* is a learning paradigm where a collection of a finite number of DNNs is trained for the same task. In general, *deep ensembles* is constructed in two steps: (1) training a number of DNNs in parallel without any interaction, and (2) calculating the weighted average of the estimation results of each DNN as the final output of the *deep ensembles*.

### 4.4.1 Cardinality Transformation

In this section, we discuss how to create proper training labels through transformations. We generate a set of queries (§4.4.2) $\mathcal{S} = (q_1 : Act(q_1)), ..., (q_N : Act(q_N))$ with actual cardinality as the label, where $\mathcal{S}$ contains $N$ labeled queries. The cardinality variation across different queries in $\mathcal{S}$ can be huge, and the distribution of the actual cardinalities for different queries can be skewed. Building an accurate model on such data is challenging. We alleviate this problem by normalizing the actual cardinalities in $\mathcal{S}$ before training (i.e., the normalized values of the actual cardinalities belong to $[0, 1]$).

We use the *log transformation* and *min-max scaling* to do the transformation. At runtime when using Fauce for estimation, we apply inverse transformation to get

the true estimations.

**Log transformation.**The log transformation allows the model $\mathcal{M}$ to capture the abrupt variation. We apply log transformation (using the base 2) on the cardinalities to mitigate such variation.

**Min-max scaling.** We rescale the outcomes of the *log transformation* into the range $[0, 1]$ using *min-max scaling*. Given a set of log transformed cardinalities CARD $= \{card_1, card_2, ..., card_n\}$, the max cardinality in CARD ($max_{card}$), the min cardinality in CARD ($min_{card}$), the result of the *min-max scaling* for $card_i$ in CARD is calculated as, $card_i^{'} = \frac{card_i - min_{card}}{max_{card} - min_{card}}$. Therefore, the final cardinality estimation is formulated by inverse transformation, which is represented as: $est(q_i) = 2^{card_i^{'} \times (max_{card} - min_{card}) + min_{card}}$.

### 4.4.2 Training Data Generation

Since the distribution of the cardinalities for the queries can be easily skewed, naive sampling from the space of all queries can result in a highly non-uniform training dataset and a sub-optimal cardinality estimator. In order to generate a uniform training dataset, our training data generation uses the following two rules: (1) **Generality.** The queries should come from different *join graphs* derived from the *join schema* of the database; (2) **Diversity.** The training data of the queries should be diverse in the number of predicates and their cardinalities. Based on these two rules, our training data is generated as follows.

We make the generated queries uniformly distributed to each *join graph*. To generate a query to a *join graph*, we first draw a tuple from the inner join result and get the number of non-null columns of this tuple, denoted as $N_c$. Second, we choose the number of predicates $n_p \in \{2, 3, ..., N_c\}$ uniformly at random. Then we randomly choose $n_p$ columns, and randomly place $n_p$ comparison operators associated with these columns based on whether each column can support range ($\{\leq, \geq, =\}$) or equality filters ($=$). These two steps guarantee a diverse set of multi-predicate queries.

## 4.5 Model Design

Fauce includes two complementary approaches that operate in two phases, shown in Figure 4.4. We first train the model offline, then we use the trained model for the inference online. The outcome of the inference includes, estimated cardinality and both model uncertainty and data uncertainty. $Card$ denotes the estimated cardinality; $U_m$ is the model uncertainty with respect to the estimation; $U_m$ is the query-dependent data uncertainty. Queries with high uncertainties are used for the incremental learning. Dotted neurons represent Dropout.

In the offline phase, we train the model $\mathcal{M}$ based on the generated training data (Section 4.4.2); In the online phase, the model $\mathcal{M}$ accepts queries and outputs their estimated cardinalities together with the estimation uncertainties. The training first generates a set of labeled queries $\mathcal{S}$ (§4.4.2). Then we apply *query featurization* (§4.3) and *cardinality transformation* (§4.4.1) on $\mathcal{S}$ to get the training dataset $D$. $D$ consists of $N$ featurized queries $\{x_i, y_i\}_{i=1}^{N}$, where $x_i \in \mathbf{R}^L$ represents the L-dimensional query features, and $y_i \in \mathbf{R}$ is a real value in the range of $[0, 1]$. Let $K$ denote the number of DNNs in the deep ensemble, and $W = \{w_i\}_{i=1}^{K}$ denote the parameters of the ensemble where $w_i$ is the parameters of a DNN. Once the training is finished offline, the parameters $W$ will be used for inference online.

A query for inference is featurized as a real valued vector, and then this vector is plugged into the trained model to estimate the cardinality and uncertainty of this estimation. The uncertainty consists of *model uncertainty* and *data uncertainty*, where the model uncertainty describes how confident the learned model is, and the data uncertain measures how noisy the collected query data are. These two types of uncertainty values will be leveraged to boost the model accuracy. The high model uncertainty means the learned parameters $W$ cannot best describe the distribution of the features of a query. In this case, this query will be collected for the future retraining. The high data uncertainty means the noisy of the query data (e.g., new updated data in database) is high. In this case, we generate a bunch of new training queries based on this query with the high data uncertainty by a sampling method [67], and use these queries for the future training. That is, we use an *incremental learning* strategy to boost the model accuracy.

### 4.5.1 Uncertainty Quantification

Model uncertainty can be quantified using the Bayesian neural network [68, 69] (BNN) that captures uncertainty about the learned parameters. Data uncertainty describes the shift between the generated training data and input queries. To quantify the uncertainty, we use the following definition of the total variance in each estimated cardinality, based on [70]. Assuming $x$ is the feature vector of the query $q$, $y$ is $q$'s estimated cardinality before inverse-transformation, the variance in $y$ is formulated as follows.

$$Var(y) = Var(\mathbf{E}[y|x]) + \mathbf{E}[Var(y|x)] \tag{4.1}$$

Based on Equation 4.1, we define the model uncertainty and data uncertainty as follows.

$$U_m(y|x) = Var(\mathbf{E}[y|x]) \tag{4.2}$$

$$U_d(y|x) = \mathbf{E}[Var(y|x)] \tag{4.3}$$

where $U_m$ and $U_d$ represent the model and data uncertainty respectively. We can see that both uncertainties explain the variance in the estimation. The model uncertainty explains the variance related to $\mathbf{E}[y|x]$, and the data uncertainty explains the variance inherent to the conditional distribution $Var(y|x)$.

**Model uncertainty.** BNNs are used to find the posterior distribution of parameters $W$ for Fauce, given the dataset $D = \{x_i, y_i\}_{i=1}^{N}$. Assume that the posterior distribution of $W$ is $p(W|D)$, and $f_W = \{f_{w_i}\}_{i=1}^{N}$ is the function mapping for the deep ensembles between $\{x_i\}_{i=1}^{N}$ and $\{y_i\}_{i=1}^{N}$. Given an inference query $q^*$, its feature vector is $x^*$. The estimated cardinality is calculated by marginalizing over the posterior distribution, shown as follows.

$$p(y^*|x^*, D) = \int_W p(y^*|f_W(x^*))p(W|D)dW \tag{4.4}$$

In Equation 4.4, $y^*$ is the estimated cardinality for the query $q^*$ before the inverse-transformation. Here, the exact computation for $p(W|D)$ is intractable, so we use a variational inference method [71] to find an approximation $q(W)$ to the poste-

rior distribution $p(W|D)$. The estimation distribution is approximated by switching $p(W|D)$ to $q(W)$ in Equation 4.4 and performing the Monte Carlo integration, $\mathbf{E}(y^*|x^*) \approx \frac{1}{K}\sum_{i=1}^{K} f_{w_i}(x^*)$. The predictive variance can also be approximated as, $Var(y^*) \approx \frac{1}{K}\sum_{i=1}^{K} f_{w_i}(x^*)^2 - \mathbf{E}(y^*|x^*)^2$. $Var(y^*)$ arises because of the uncertainty about the model parameters $W$. We use $Var(y^*)$ to quantify the model uncertainty in Fauce.

**Data uncertainty.** Data uncertainty is dependent on the input queries. We need a model that not only estimates the output cardinalities, but also estimates the variances of the cardinalities given the input queries. That is, the model must give an estimation of $Var(y|x)$ mentioned in Equation 4.3. Assume that $\mu(x)$ and $\sigma(x)$ are the functions parameterized by $W$ that calculate the mean and standard deviation of the estimation for a query $q$ respectively, and $x$ is $q$'s feature vector. We have $y \sim \mathcal{N}(\mu(x), \sigma(x)^2)$, and the negative log likelihood is written as follows.

$$Loss(W) = \frac{1}{N}\sum_{i=1}^{N}(\frac{\log \sigma^2(x_i)}{2} + \frac{(y_i - \mu(x_i))^2}{2\sigma^2(x_i)} + \frac{1}{2}log2\pi) \tag{4.5}$$

Comparing Equation 4.5 with a standard mean squared loss used in the traditional regression, we can see that the ensemble model introduces higher estimation variances for queries where the mean of estimated cardinality $\mu(x_i)$ is more deviated from the true cardinality $y_i$. On the other hand, a regularization term on $\sigma(x_i)$ prevents the model from introducing high estimation variances for all queries. After the model is optimized, we use $\sigma^2(x^*)$ to estimate the data uncertainty of a new query $q^*$, where $x^*$ is the feature vector after $q^*$ is featurized.

### 4.5.2 Training and Inference

**Ensembles training.** Fauce uses the entire training dataset $D$ to train each DNN. To improve the model's robustness, Fauce also includes the *adversarial training*. In particular, we use the fast gradient sign method [72] to generate adversarial query examples. Given a query $q$, $x$ as $q$'s feature vector, and $y$ as the query's true cardinality, an adversarial example is generated by $x^{'} = x + \eta sign(\bigtriangledown_x Loss(W, x, y))$, where Loss(W, x, y) is from Equation 4.5. Here, $\eta$ is a small value to bound the max perturbation. Those adversarial examples generated by the above formulation are

used to augment the original training set $D$ by treating $(x^{'}, y)$ as additional training examples.

**Ensembles inference.** We treat the ensemble as a uniformly-weighted mixture model to calculate the final estimation results. Assume that $x^*$ is the feature vector of the query $q^*$. The estimated cardinality of $q^*$ is calculated as

$$Card(q^*) = \mathbf{E}(y^*|x^*) \approx \frac{1}{M}\sum_{i=1}^{M}\mu_{w_i}(x^*) \tag{4.6}$$

The model uncertainty for the query $q^*$ is measured as,

$$U_m(q^*) = \frac{1}{M}\sum_{i=1}^{M}\mu_{w_i}(x^*)^2 - \mathbf{E}(y^*|x^*)^2 \tag{4.7}$$

and the data uncertainty for the query $q^*$ is quantified as,

$$U_d(q^*) = \frac{1}{M}\sum_{i=1}^{M}(\sigma^2_{w_i}(x^*) + \mu^2_{w_i}(x^*)) - \mathbf{E}(y^*|x^*)^2 \tag{4.8}$$

### 4.5.3 Management of Estimation Uncertainty

**Uncertainty management.** We propose an algorithm called *manage uncertainty* (Algorithm 4) to use the uncertainties to make the estimation safer to use and improve model accuracy. In Algorithm 4, $\phi_m$ and $\phi_d$ are two thresholds for the model uncertainty $U_m(q^*)$ and data uncertainty $U_d(q^*)$ respectively.

When comparing the uncertainty values with the thresholds, we have three situations. First, $U_m(q^*) \leq \phi_m$ and $U_d(q^*) \leq \phi_d$. This means that Fauce is confident on its estimation, so the estimated cardinality is safe to use (Lines 2-3). Second, $U_m(q^*) > \phi_m$ and $U_d(q^*) \leq \phi_d$. This happens when the training dataset $D$ well represents the features of $q^*$, but the trained parameters underestimate $q^*$. We store the query $q^*$ into a buffer $B$ for an *incremental learning* strategy to eliminate the underestimation (Lines 4-5). Third, $U_m(q^*) > \phi_m$ and $U_d(q^*) > \phi_d$. This happens when the training dataset $D$ cannot represent the features of the query $q^*$, and the parameters underestimate $q^*$. Besides storing the query $q^*$ into the buffere $B$, we enlarge the number of queries in $B$ by sampling [67] additional training data based on $q^*$ for the incremental learning (Lines 6-9). At last, we update the model $\mathcal{M}$ (Line

10).

**Algorithm 4** ManageUncertainty($U_m(q^*), U_d(q^*), \phi_m, \phi_d$)

**Input:** $U_m(q^*)$: Model uncertainty for new query $q^*$
$U_d(q^*)$: Data uncertainty for new query $q^*$
$Card(q^*)$: Estimated cardinality for $q^*$
$\phi_m$, $\phi_d$: Threshold for the model, data uncertainty
**Output:** $Safe\_Card$: Cardinality that is safety to use
**Output:** $\mathcal{M}^*$: updated model $\mathcal{M}$
$B = \{\}$; `// Buffer to store queries for retraining`
**if** $U_m(q^*) \leq \phi_m$ ***and*** $U_d(q^*) \leq \phi_d$ **then**
  $Safe\_Card := Card(q^*)$
**else if** $U_m(q^*) > \phi_m$ ***and*** $U_d(q^*) \leq \phi_d$ **then**
  $B = B \oplus q^*$
**else if** $U_m(q^*) > \phi_m$ ***and*** $U_d(q^*) > \phi_d$ **then**
  $B = B \oplus q^*$ $B \cup =$ Sampling($q^*$); `// Sampling queries[67]`
$\mathcal{M}^* \leftarrow$ IncrementalLearning($\mathcal{M}, B$) **return** $Safe\_Card, \mathcal{M}^*$

**When is the incremental learning triggered?** In Algorithm 4, the incremental learning can be triggered in two cases. In the first case, the incremental learning is triggered when the number of queries in $B$ is beyond $B$'s maximal size. In Fauce, we set the maximal size of $B$ as 2000 queries. A small maximal size of $B$ can frequently trigger the incremental learning, which increases the overhead of using the incremental learning. In contrast, a large maximal size of $B$ may rarely trigger the incremental learning, which means that a large number of queries will be estimated by a stale model. As a consequence, the estimation quality of Fauce is decreased. In the second case, the incremental learning is triggered when a large fraction of estimated queries with high uncertainties happen. This scenario usually happens in a dynamic environment. As data are continuously updated, the workload and data distribution shift may happen. The learned estimator can no longer accurately represent the distribution of the updated data. As a result, the queries based on new updated data have high uncertainties.

**Re-encoding for tables/joins/columns.** The tables and joins encoding is based on the join schema of a database. Thus, re-encoding of them is not necessary when incremental learning happens in both static and dynamic environments. The columns encoding is based on the inter-column correlations. Such correlations do not change in a static environment. Thus, re-encoding of columns is not necessary. However, in a dynamic environment, the inter-column correlations could change when

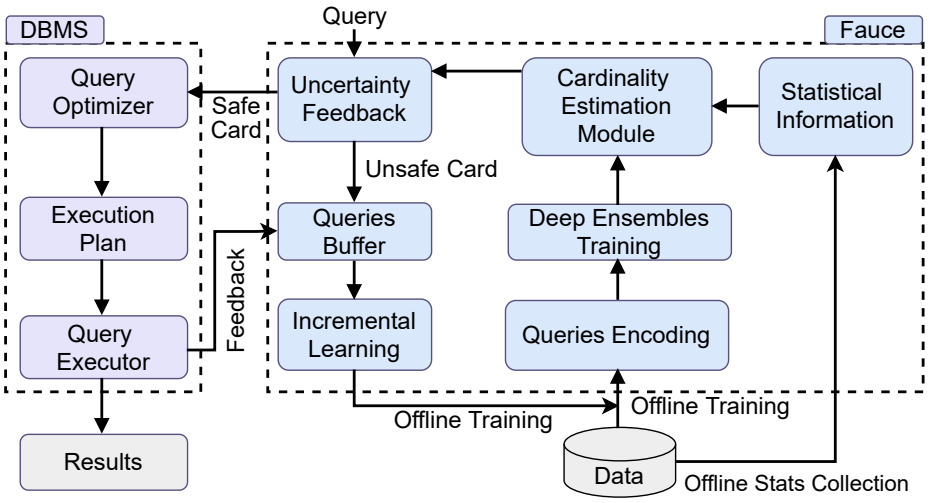


**Figure 4.5:** Integration of Fauce with existing DBMS

data are continuously updated. However, Fauce does not need to re-encode of all the columns from scratch. First, Fauce calculates the inter-column correlations only based on the new coming data. Then, Fauce filters out those pairwise columns whose correlations are significantly changed, and only re-encodes those columns. Therefore, re-encoding of columns only happens on a portion of columns in a dynamic environment. In Fauce, columns re-encoding can be finished within 50 seconds on our testing workloads in a dynamic environment.

### 4.5.4 Integration with DBMS

Figure4.5 shows how Fauce is integrated into a DBMS. Fauce is performed before the query optimizer as an additional phase that estimates the cardinalities and uncertainties. If the estimated uncertainties are less than the threshold $\phi_m$ and $\phi_m$, then the cardinalities are injected into the query optimizer. In Fauce, the cardinality estimation techniques have online and offline phases. The offline phase includes three components: (1) statistical information collection; (2) deep ensembles training; and (3) incremental learning. These three components happen in both static and dynamic scenarios. But in a dynamic scenario, the statistical information can be outdated as new data are continuously coming. Thus, Fauce must update the statistics in the dynamic scenario. In dynamic scenario, the re-encoding of columns is required when incremental learning happens. The updated data may change the correlations among columns. Thus, Fauce must update the global column-dependency graph(§4.3.2) for

the database. The online phase is for inference. This phase is the same in both static and dynamic scenarios. Before inference, a query must be featurized into vectors. As Fauce has received the statistics information about the database tables and encoding results of all the tables, joins, and columns in the offline phase, the time overhead for query featurization is small, which usually takes 2-6ms in our evaluation.

**How is the incremental learning tied to the database system?** Figure 4.5 shows that the incremental learning is tied to the database system in two ways. For the first way, we can get uncertainty feedback when use Fauce to estimate the cardinalities. The queries with large estimated uncertainties will be stored in a buffer $B$ for the offline incremental learning. For the second way, we can directly use feedback from the query executor for incremental learning. However, the incremental learning based on new queries could affect the existing queries. In other words, the model "forgets" the old data and focuses exclusively on the new data. We use the Dropout [73] technique to avoid the above problem. In Fauce, we utilize a dropout value of $p = 0.2$ when updating the model $\mathcal{M}$ over the queries in $B$. Here, the overhead of the incremental learning is around 2 mins.

## 4.6 Evaluation

We compare Fauce with state-of-the-art cardinality estimators using point and range queries. We aim to answer the following questions:

- Compared with the prior methods, how does Fauce perform in terms of accuracy and efficiency? (§4.6.2 and §4.6.4)
- How does the improvement on the cardinality estimation impact the performance of the query optimiser (§4.6.3)
- How does the Fauce perform in a dynamic environment? (§4.6.5)
- How does Fauce compare with other literature on data profiling? (§4.6.7)

### 4.6.1 Experimental Setup

**Platform.** We use a machine with an NVIDIA V100 GPU and an Intel i9 CPU with 128GB RAM, and Tensorflow 2.3.

**Table 4.2:** Workloads used for evaluation.

| Workload | Tables | Rows | Cols | Feature |
|---|---|---|---|---|
| JOB-base | 6 | $2 \cdot 10^{12}$ | 13 | normal queries |
| JOB-more-filters | 6 | $2 \cdot 10^{12}$ | 22 | + more filters |
| JOB-complex-joins | 15 | $2 \cdot 10^{13}$ | 22 | + complex joins |

**Workloads.** We use a real-world dataset: IMDB [23]. IMDB has complex correlated columns. It consists of 21 tables. We use the same method in training data generation (§4.4.2) to generate testing queries. In our experiments, each workload contains 2000 testing queries. Those workloads are discussed as follows (see Table 4.2). In Table 4.2, *Tables*: Number of base tables in each workload. *Rows*: Number of rows after the outer join. *Cols*: Total number of columns in the base tables. *Feature*: Characteristic of the queries in each workload.

- **JOB-base**: the queries in JOB-base are generated based on numeric columns in JOB-light. The schema in JOB-light is a typical star schema. JOB-light contains six tables, `title` (primary), `cast_info`, `movie_info`, `movie_companies`, `movie_keyword`, and `movie_info_idx`. The predicates of the queries have 2-7 filters.

- **JOB-more-filters**: this benchmark tests Fauce's scalability to complicated predicates. Some queries involve large number of columns in their predicates. The schema is the same as JOB-base's. The predicates of the testing queries have 8-13 filters.

- **JOB-complex-joins**: this benchmark contains 15 tables in IMDB and involves multiple join keys. For instance, `movie_companies` is not only joined with `title` on `movie_id`, but also joined with `company_name` on `company_id`, etc. Each query joins 2–11 tables. JOB-complex-joins is used to test Fauce's scalability to complicated join conditions.

**Evaluation metrics.** To evaluate the accuracy of Fauce on the above workloads, we use the q-error metric. The q-error of Fauce on a query $q$ is calculated as, $error = max(\frac{est(q)}{act(q)}, \frac{act(q)}{est(q)})$. Here, we assume that $act(q) \geq 1$ and $est(q) \geq 1$, so the minimum $error$ is $1\times$. We report the median, $75th$, $90th$, $95th$ and $99th$ percentile errors across all queries.

**Table 4.3:** Estimation errors on the JOB-base, JOB-more-filters, JOB-complex-joins workloads.

| Estimator | JOb-base | | | | | JOB-more-filters | | | | | JOB-complex-joins | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 50th | 75th | 90th | 95th | 99th | 50th | 75th | 90th | 95th | 99th | 50th | 75th | 90th | 95th | 99th |
| Postgres | 13.4 | 623 | 1960 | $2 \cdot 10^5$ | $7 \cdot 10^5$ | 8.1 | 162 | 1429 | $1 \cdot 10^4$ | $2 \cdot 10^5$ | 17.4 | 1679 | 7928 | $4 \cdot 10^5$ | $8 \cdot 10^5$ |
| IBJS | 11.6 | 125 | 2321 | $4 \cdot 10^5$ | $7 \cdot 10^6$ | 7.6 | 77.1 | 963 | $8 \cdot 10^3$ | $4 \cdot 10^5$ | 14.5 | 239 | 5014 | $3 \cdot 10^5$ | $7 \cdot 10^5$ |
| MSCN | 6.13 | 44.5 | 142 | 3568 | $2 \cdot 10^4$ | 4.8 | 16.3 | 121 | 1680 | $5 \cdot 10^4$ | 6.9 | 34.4 | 163 | 2820 | $4 \cdot 10^4$ |
| DeepDB | 4.61 | 17.3 | 145 | 3348 | $3 \cdot 10^4$ | 4.2 | 14.5 | 86 | 1182 | $4 \cdot 10^4$ | 5.3 | 17.3 | 268 | 3717 | $3 \cdot 10^4$ |
| NeuroCard | 4.15 | 10.2 | 107 | 1693 | $9 \cdot 10^3$ | 3.8 | 8.2 | 59 | 538 | $2 \cdot 10^4$ | 4.5 | 8.8 | 56.7 | 608 | $8 \cdot 10^3$ |
| Fauce+DU | 4.12 | 9.6 | 46.5 | 1246 | $8 \cdot 10^3$ | 3.9 | 7.7 | 43 | 464 | $1 \cdot 10^4$ | 4.1 | 8.4 | 48.0 | 598 | $9 \cdot 10^3$ |
| Fauce+MU | 2.86 | 5.5 | 17.2 | 375 | $3 \cdot 10^3$ | 3.2 | 5.6 | 25 | 245 | $5 \cdot 10^3$ | 3.4 | 7.2 | 25.8 | 366 | $5 \cdot 10^3$ |
| Fauce+Both | **2.58** | **5.1** | **15.3** | **279** | $\mathbf{2 \cdot 10^3}$ | **2.9** | **5.1** | **21** | **206** | $\mathbf{3 \cdot 10^3}$ | **2.7** | **6.3** | **21.6** | **227** | $\mathbf{3 \cdot 10^3}$ |

**Baselines.** We compare Fauce against a variety of representative cardinality estimators, including:

1) Postgres: Using Postgres, we evaluate the cardinality estimation that can be obtained from a real DBMS. The cardinality estimation in Postgres relies on 1D histograms and heuristics.

2) IBJS: We use the Index-based Join Sampling method (IBJS) [21] as a non-learned baseline. IBJS estimates a query's cardinality using a sampling-based approach based on the query's join graph and executing per-table filters.

3) MSCN: This is a representative supervised query-driven estimator [16]. We generate 10K training queries for each workload to train the model and use a bitmap size of 2K.

4) DeepDB: This is an unsupervised data-driven estimator [17]. DeepDB uses a non-neural sum-product network as the density estimator for each table subset chosen by correlation tests. Conditional independence is assumed across subsets.

5) NeuroCard: This is also an unsupervised data-driven estimator [19]. NeuroCard is a join cardinality estimator that builds a single neural density estimator over the entire database.

### 4.6.2 Estimation Quality

Tables 4.3 shows that Fauce exceeds the baseline estimators on all the three workloads (§ 4.6.1). "MU" and "DU" denote model uncertainty and data uncertainty respectively. "Fauce+MU" means training with model uncertainty only, "Fauce+DU" means training with data uncertainty only, and "Fauce+Both" means training with

both model uncertainty and data uncertainty.

**(1) Results on JOB-base.**

**Postgres has the largest median, 75th, and 90th error.** Postgres only relies on 1D histogram and heuristics, and does not contain cross-column statistics. Thus, Postgres cannot fully capture the characteristics of queries, and has high estimation error.

**IBJS has the largest 95th and 99th errors.** IBJS is a sampling based method. We set the maximum sampling budget as 10,000, as a larger sampling budget does not bring too much benefit [21]. Because the joint space is very large, those samples have small chances to hit testing queries, hence can cause large estimation errors. IBJS's inference time varies from 3 to 20 ms. For queries with many joins, IBJS must store hundreds or even thousands of intermediate results. As a consequence, IBJS's memory size varies from 40 KB to 4 MB.

**MSCN has large estimation errors on some queries with small true cardinalities.** MSCN's training is based on a number of featurized queries. MSCN does not contain uncertainty information about testing queries. Furthermore, its query featurization method cannot leverage semantic information in a database. As a result, MSCN has large errors on some queries.

**DeepDB has large errors on high quantiles.** DeepDB uses a sum-product network to estimate the density for each table subset. Each table subset is chosen based on the correlations among tables in the database. DeepDB assumes conditional independence across table subsets. But this assumption is not always true in real databases as some table subsets may have close relationships. Furthermore, DeepDB assumes inter-column independence when building the density model via the sum-product network. Therefore, it does not reflect real column dependencies in the databases. Because of the above assumptions, DeepDB has limited expressiveness, which causes large errors on high quantiles. As a result, Fauce's accuracy gain on each quantile is 1.8×, 3.4×, 9.5×, 12×, and 15×, compared with DeepDB.

**Fauce exceeds NeuroCard.** NeuroCard uses deep autoregressive models as a density estimator to learn high-dimensional data distributions. It works as follows. Given a range query with $K$ predicates, first, NeuroCard obtains the probability of $i$-th predicate conditioned on previous values. Then, it generates a sample value for $i$-th column. Finally, the conditional probabilities are multiplied together to estimate

the cardinality. We find NeuroCard tends to have large errors on some range queries with correlated columns in their predicates. In contrast, Fauce is robust to this kind of queries. The overall Fauce's accuracy gain on each quantile is 1.6×, 2×, 7×, 6.1×, and 4.5×, compared with NeuroCard.

**(2) Results on JOB-more-filters.** This workload is used for testing Fauce's scalability on queries with a large number of filters in their predicates. As Table 6.4 shows, all estimators produce less accurate cardinalities than Fauce. Compared with Postgres, Fauce's accuracy gain is from 2.8× to 70×, because the accumulative error caused by the 1D histogram grows as the number of filters grows. Compared with IBJS, Fauce's accuracy gain is from 2.6× to 46×. This is because the sampling results can easily be empty as the number of filter grows. Compared with MSCN, Fauce's accuracy gain is 1.7×, 3.2×, 5.9×, 8.1× and and 16.7× at median, 75th, 90th, 95th, and 99th respectively. Compared with DeepDB, Fauce improves the accuracy by 1.5×, 2.8×, 4×, 5.7×, and 13.3× at median, 75th, 90th, 95th, and 99th respectively. At last, compared with NeuroCard, Fauce's accuracy gains is 1.3×, 1.6×, 2.8×, 2.6× and 6.7× at median, 75th, 90th, 95th, and 99th respectively. Existing estimators fail to capture the more complex inter-column correlations. As a result, their estimations are vulnerable to queries with a large number of columns in predicates. The results demonstrate Fauce's scalability to the number of filters.

**(3) Results on JOB-complex-joins.**

This workload is used for testing the Fauce's ability to scale to queries with a large number of filters and multiple join keys. The number of filters in the predicates of the queries varies from 2 to 13; The possible number of join keys varies from 2 to 10, and the predicates of the queries can have multiple join keys. Table 6.4 shows that Fauce's accuracy remains high on this complex schema. Postgres and IBJS have the largest errors, because many intermediate samples become empty. Compared with MSCN and DeepDB, Fauce's accuracy improvement is up to 13.3× and 10× respectively. NeuroCard also achieves high accuracy, but it still has large estimation errors on queries with correlated columns in predicates. Fauce overcomes this challenge and offers better accuracy than NeuroCard.

**(4) Results on queries with no joins.** We use the methods in [65] as the baseline. The estimators in [65] are based on light-weight models (i.e., simple NNs and boosting trees). We refer to the methods using NNs and boosting trees in [65]

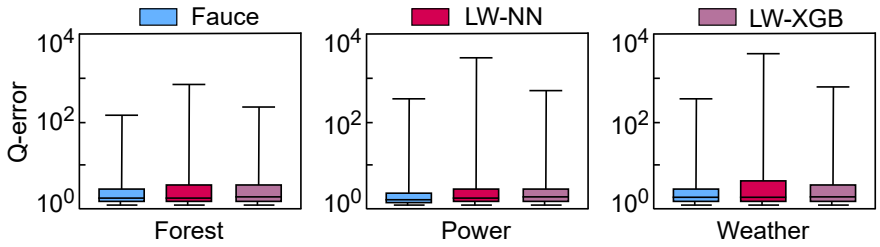


**Figure 4.6:** Estimation errors on various datasets with no joins.

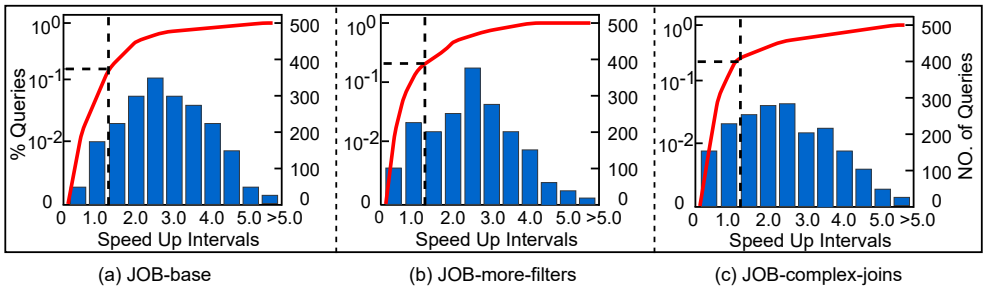


**Figure 4.7:** The impact of the improved cardinality estimation of Fauce on query performance.

as "LW-NN" and "LW-XGB" respectively. Our experiments use the same datasets as [65], including "Forest", "Power", and "Weather". Fauce, LW-NN, and LW-XGB are trained and tested on the same queries. Figure 4.6 shows the testing results. We can see Fauce has the higher accuracy than LW-NN and LW-XGB on all the datasets. The main difference between Fauce and LW-NN lies in the *query featurization* method. LW-NN and LW-XG extract features from $\langle Values \rangle$ of queries, and use AVI [74], EBO[75], and MinSel[76] as extra features during *query featurization*. Fauce, besides extracting features from $\langle Value \rangle$, uses the *Columns2vec* algorithm to extract features from $\langle Column \rangle$ of a query. The higher accuracy of Fauce on these datasets indicates that Fauce's featurization method can capture more informative features of a query than the query featurization method used in [65].

### 4.6.3 Impacts on Query Performance

We evaluate whether the improvement of cardinality estimation in Fauce leads to better query performance. Our evaluation is based on the workloads introduced in Section 4.6.1. We test 2000 queries for each workload. After we get the estimated cardinalities from Fauce, these estimations are then fed into a version of PostgreSQL

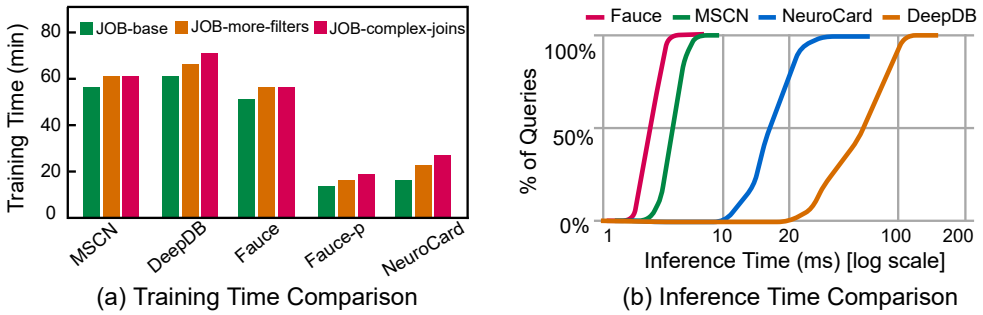


**Figure 4.8:** Physical efficiency of Fauce.

modified to accept external cardinality estimations [77]. Figure 4.7 shows the performance impact of the cardinalities estimated by Fauce, compared to the default cardinality estimations from PostgreSQL. For the Job-base (Figure4.7(a)), the execution time for these queries ranges from $<$ 1s up to 200s. Fauce improves the performance of 81.4% of the queries. For the Job-more-filters (Figure4.7(b)), the majority of the queries' runtime ranges from 0.5 to 350s. Fauce improves the performance for 80.3% of the queries. 8.4% of the queries' execution time is extended, and the rest of queries have the same performance as PostgreSQL. For the Job-complex-joins (Figure4.7(c)), Fauce improves the performance for 78.2% of the queries.

### 4.6.4 Efficiency of Fauce

**Training time comparison.** Figure 4.8(a) shows the training time. Once the training queries are collected, training MSCN takes 55-60 mins for the three workloads. DeepDB can only run on parallel CPUs (not on GPU as other methods), so DeepDB takes the longest training time (60-75 mins). Fauce has higher efficiency on *query featurization* compared with MSCN, so training Fauce requires less time than MSCN: Fauce takes 50-54 mins on all the three workloads. Note that DNNs in the ensemble are independent in Fauce and can be trained independently. Therefore, Fauce's training time can be further optimized through parallel training, which reduces the training time to less than 20 mins (see Fauce-p in Figure 4.8(b)). Training NeuroCard has to calculate the join count tables and perform parallel sampling first, and then trains the auto-regressive model for some epochs. Even if training process of NeuroCard is accelerated by GPUs, training Neurocard still takes more than 20 mins on Job-more-filters and Job-complex-joins workloads. Our methods Fauce-p takes

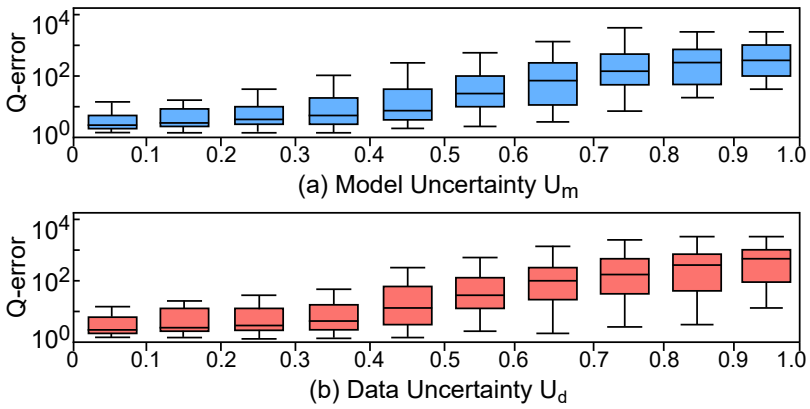


**Figure 4.9:** Q-error of queries with different uncertainties.

the shortest training time.

**Inference time comparison.** Figure 4.8(b) shows the inference time of MSCN, DeepDB, NeuroCard, and Fauce on JOB-base workload. MSCN, NeuroCard, and Fauce run on GPU while DeepDB runs on CPU; These estimators are implemented in Python. Fauce and MSCN are the fastest because they are based on lightweight network and involve fewer calculation during the inference. DeepDB's inference time spans from 1 ms to 200ms, and its inference time is short for queries with a small number of joins and filters. However, its inference time can be more than 150ms for complex queries. NeuroCard's inference time is smaller than DeepDB, but it is still 2 to 10× larger than those of Fauce and MSCN. This is due to a large number of floating point operations involved in the autoregressive model of NeuroCard. The inference time of DeepDB and NeuroCard is more sensitive to the complexity of the queries than Fauce and MSCN.

### 4.6.5 Handling Data Updates

We analyze how Fauce performs in a dynamic environment. It is common that data are continuously updated in databases.

**Threshold values.** In Algorithm 4, we use two thresholds $\phi_m$ and $\phi_d$ to control the model uncertainty and data uncertainty respectively. Here, we discuss how we set proper values for $\phi_m$ and $\phi_d$ as the thresholds. In our study, the threshold values for $\phi_m$ and $\phi_d$ are measured based on additional 10K queries, not those for testing. Those queries are derived from JOB-light. We estimate cardinalities and uncertainties for

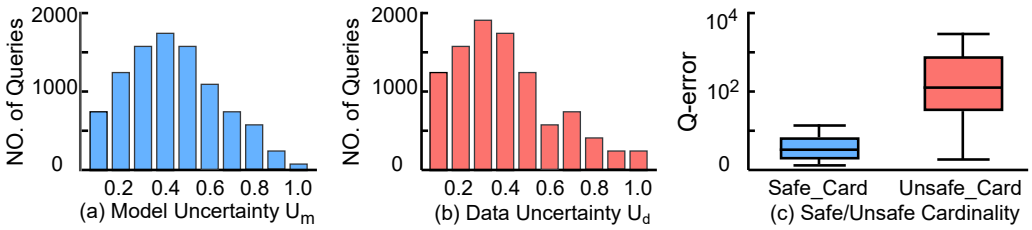


**Figure 4.10:** Statistical information for the queries.

those 10K queries. The uncertainty value of each query is in the range of [0,1]. We set the length of a uncertainty interval as 0.1 and use ten uncertainty intervals. We count the number of queries in each of the ten uncertainty intervals. Figure 4.9(a) and (b) show that queries with $\phi_m$ higher than 0.5 or $\phi_d$ higher than 0.4 tend to have large errors. Based on the above observation, we set the threshold value for model uncertainty and data uncertainty as 0.5 and 0.4 respectively. In Fauce, if a query's model uncertainty is below $\phi_m$ or a query's data uncertainty is below $\phi_d$, then its estimated cardinality is safe to use. We refer to such cardinality as "safe_card". Figure 4.10(a) and (b) show the number of queries within the ten intervals. We can see the percentage of safe_card is about 70%. The errors of queries within safe_card and unsafe_card are shown in Figure 4.10(c), based on which we conclude that the queries with safe_card have much smaller errors than with unsafe_card.

**Dynamic environment setup.** Suppose that there are $n$ queries uniformly distributed in a time range $[T_i, T_{i+1}]$, and $T = T_{i+1} - T_i$. The queries based on updated data begin to come at timestamp $T_i$. Those queries with high uncertainties are stored in the buffer $B$ for the incremental learning. Once the number of queries in the buffer $B$ is beyond $B$'s maximum capacity, the model update begins. Suppose the model update finishes at timestamp $T_f$ ($T_i < T_f \leq T_{i+1}$). For the first $\lceil n \cdot \frac{T_f - Ti}{T} \rceil$ queries, their cardinalities are estimated using the stale model $\mathcal{M}_{stale}$. For the remaining $\lfloor n \cdot (1 - \frac{T_f - Ti}{T}) \rfloor$ queries, the updated model $\mathcal{M}_{update}$ are used. Since some queries are handled by the (inaccurate) stale model, the estimation results for these queries can be erroneous.

**Data update.** We use the real-world dataset IMDB [23] for testing under a dynamic environment. Our experiment is based on two different kinds of data updates. The first kind of data updates leads to significant changes in pair-wise correlations, while the second kind of data updates does not. (a) In the first kind

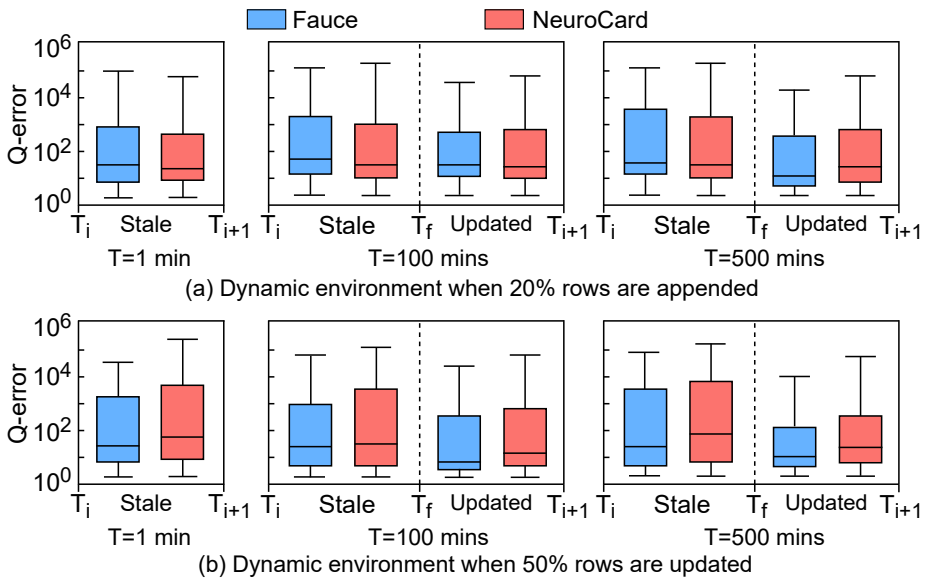


**Figure 4.11:** Estimation quality under dynamic environment.

of data update, we use the similar method introduced in [65] to update the dataset. In particular, we update 50% tuples of the dataset, which results in huge change in data distribution; (b) In the second kind of data update, we partition the table `title` into two parts on the `year` column. The part with the latest `year` is used as the new data to be appended into dataset, the pair-wise correlations for this method are not significantly changed. This kind of data update is used in [19]. After data updates, we apply our workload generation method on the updated dataset to generate 10K queries for testing. These queries are uniformly distributed in $[T_i, T_{i+1}]$. $T$, which is equal to $T_{i+1} - T_i$, is a parameter, which represents how "frequently" the data are updated. For example, if the data are periodically updated every 100 mins, then $T$ is 100 mins.

**Model update.** We update Fauce and NeuroCard, and then compare their estimation quality. NeuroCard is a dat-driven estimator, so NeuroCard is updated by retraining on the entire new updated dataset. Fauce is a query-driven estimator. We assign 2K queries for Fauce, as 2K is the maximum capacity of the buffer $B$ to store queries for the incremental learning.

**Estimations in a dynamic environment.** We test the estimation quality of Fauce and NeuroCard in a dynamic environment. The value of $T$ is varied to control the frequency of data update. We set three levels of the frequency: high (1 min), medium (100 mins), and low (500 mins). The estimation quality of NeuroCard and

Fauce in the dynamic environment is shown in Figure 4.11.

First, we compare Fauce with NeuroCard when 20% of rows in the table `title` is appended (Figure4.11(a)). If the frequency of the data update is high (shown in the left figure in Figure4.11(a)), both Fauce and NeuroCard cannot finish the model update, then the stale models for Fauce and NeuroCard are used for testing. When the data update does not change the data distribution, data distribution learned by NeuroCard still works. As a result, NeuroCard performs better than Fauce. When the data update frequency is medium and slow (the right two figures in Figure4.11(a)), both Fauce and NeuroCard can finish model update. We set the time interval for data updates as $T = T_{i+1} - T_i$, for the queries coming within $[T_i, T_f]$, and those queries are tested by the stale models. Here, $T_f$ is the time when the model updates are finished. Queries coming within $[T_f, T_{i+1}]$ are tested by the updated models. For queries coming within $[T_i, T_f]$, NeuroCard performs better than Fauce. This is because the appended data does not change the data distribution in the database. So NeuroCard can still work well. Queries coming within $[T_f, T_{i+1}]$ are tested by the updated models in Fauce and NeuroCard. We can see Fauce performs better than NeuroCard.

Second, we compare Fauce and NeuroCard when 50% rows in table `title` are updated (Figure4.11(b)). Overall, Fauce performs better than NeuroCard for queries coming within $[T_i, T_f]$ (when the stale models are used) and $[T_f, T_{i+1}]$ (when the updated models are used). This is because the inter-column correlations in this scenario have been significantly changed, so the data distribution learned by NeuroCard is outdated. In Fauce, the feature vector of a query $q = \langle Tables \rangle$, $\langle Joins \rangle$, $\langle Columns \rangle$, $\langle Values \rangle$ after the *query featurization* (§4.3) has the length of $L = m\lceil \log(m+1) \rceil + n + 3C$ (see Table4.1). As the inter-column correlations have been significantly changed, the features extracted from $\langle Columns \rangle$ can not reflect the new inter-column correlations. The ratio for the features extracted from $\langle Columns \rangle$ among the total length of the feature vector is: $\frac{C}{L}$, where $C$ is the length of features extracted from $\langle Columns \rangle$. Here, the schema of the database remains the same, so the features extracted from $\langle Tables \rangle$ and $\langle Joins \rangle$ can be reused. Those features are not required for re-encoding. If the domain of some columns is changed, we need to update the domain for featurizing $\langle Values \rangle$ of a query $q$. Updating the domain of the columns can be finished in a very short time (similar with updating the 1D histogram

**Table 4.4:** Impact of encoding methods.

| Workload | Encoding | 50th | 90th | 95th | 99th |
|---|---|---|---|---|---|
| JOB-base | One-hot | 4.53 | 87 | 2029 | $3 \times 10^4$ |
| | Binary | 4.24 | 62 | 1586 | $2 \times 10^4$ |
| | **Ours** | **2.58** | **15.3** | **279** | $\mathbf{2 \times 10^3}$ |
| JOB-more-filters | One-hot | 5.23 | 84 | 862 | $2 \times 10^4$ |
| | Binary | 4.82 | 69 | 754 | $2 \times 10^4$ |
| | **Ours** | **2.9** | **21** | **206** | $\mathbf{3 \times 10^3}$ |
| JOB-complex-joins | One-hot | 5.62 | 78 | 1446 | $2 \times 10^4$ |
| | Binary | 4.77 | 62 | 1193 | $1 \times 10^4$ |
| | **Ours** | **2.7** | **21.6** | **227** | $\mathbf{3 \times 10^3}$ |

**Table 4.5:** Impact of cardinality transformation.

| Workload | Transformation | 50th | 90th | 95th | 99th |
|---|---|---|---|---|---|
| JOB-base | No | 3.22 | 32.9 | 1117 | $1 \times 10^4$ |
| | **Yes** | **2.58** | **15.3** | **279** | $\mathbf{2 \times 10^3}$ |
| JOB-more-filters | No | 3.57 | 47.4 | 429 | $2 \times 10^4$ |
| | **Yes** | **2.9** | **21** | **206** | $\mathbf{3 \times 10^3}$ |
| JOB-more-joins | No | 4.15 | 36.7 | 428 | $2 \times 10^4$ |
| | **Yes** | **2.7** | **21.6** | **227** | $\mathbf{3 \times 10^3}$ |

in DBMSs). We conclude that only the $\frac{C}{L}$ portion of features after the *query featurization* is influenced by data updates. For IMDB [23], $C$ is relatively small, compared with $L$, so Fauce's featurization method still work well in this scenario. That is why Fauce performs better than NeuroCard when 50% of rows is updated.

### 4.6.6 Other Factors Impacting Fauce

We explore how the encoding method (§4.3.1 and §4.3.2) and cardinality transformation (§4.4.1) impact Fauce's accuracy.

(1)**Impact of encoding methods.** We encode $\langle Tables \rangle$, $\langle Joins \rangle$, and $\langle Columns \rangle$ of a query with our encoding method. Some existing estimators [16, 66, 78] use one-hot or binary encoding methods. Table 4.4 shows the impact of encoding methods on the errors over the three workload (§4.6.1). This table shows the impact of different encoding methods for $\langle Tables \rangle$, $\langle Joins \rangle$, and $\langle Columns \rangle$ when *query featurization* (§4.3). "Ours" denotes our encoding method. The lowest errors are bolded. We can see the impact brought by different encoding methods for low-quantile errors is

small. However, the encoding methods have large impact on high-quantile errors. Our encoding method's accuracy gain is up to 7.3× and 15× on high-quantile errors, compared with the one-hot and binary encodings respectively.

(2)**Impact of cardinality transformation.** We use log transformation and min-max scaling (§4.4.1) to normalize the cardinalities before model training. Table 4.5 shows that the transformation reduces the error, especially the high-quantile errors. For the "Tranformation" column, value "No" means testing without transformation, and value "Yes" means testing with transformation. Lowest errors are bolded. This is because the cardinalities of queries in the training dataset are skewed, and the models directly trained on such skewed dataset have large errors. Using the transformation reduces the gap between large and small cardinalities.

### 4.6.7 Data Profiling

The column encoding method (§4.3.2) in Fauce can be used to find approximate functional dependencies (AFDs) among database columns. We compare the efficiency of Fauce with other four data profiling methods: Pyro [12], Tane [79], Ducc/Dfd [80], and Fdep [81]. Table 4.6 shows the information of the datasets we use for data profiling. The results are shown in Figure 4.12. We can see that Fauce finishes the job of finding the AFDs on all the datasets within a time limit ($10^4$s). Fauce's execution time for data profiling is the shortest on the datasets DB Status, Census, and Entity source. These datasets have unknown or large number of AFDs. When profiling on the datasets Reflns and Spots, Fauce's execution time is still lower than Tane, Ducc/Dfd, and Fdep (except for Pyro). For an easy-to-process dataset with a smaller number of rows/columns and AFDs (e.g., the Wiki image), Fauce is outperformed by the baselines. But for the hard-to-process datasets (i.e., DB Status, Census, and Entity source), the speedup of Fauce is larger than 10×, compared with baselines.

## 4.7 Summary

The cardinality estimation using machine learning models is a new research trend in the database community. However, it is challenging to make accurate cardinality estimations using machine learning models. We introduce Fauce to address this prob-

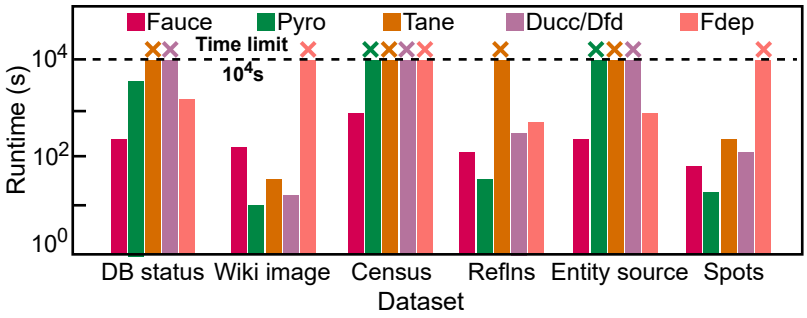


**Figure 4.12:** Runtime for data profiling. "x" means out of limit.

**Table 4.6:** Datasets used for data profiling.

| | DB status | Wiki image | Census | Reflns | Entity | Spots |
|---|---|---|---|---|---|---|
| Cols. | 35 | 12 | 42 | 37 | 46 | 15 |
| Rows | 29,787 | 777,676 | 199,524 | 24,769 | 26,139 | 973,510 |
| AFDs | 108,003 | 92 | unknown | 9,396 | unknown | 75 |

lem. Fauce has a new query featurization method which can make the input feature vectors more informative for the cardinality estimation. It also includes uncertainty information for estimation results. Experimental results show that Fauce has 1.16-91× higher accuracy than state-of-the-art solutions. By leveraging the uncertainty information, Fauce's estimation can be further improved.

# Chapter 5

# Lobster: Load Balance-Aware I/O for Distributed DNN Training

## 5.1 Overview

To address the above challenges, we propose Lobster, a holistic data loading I/O runtime for distributed DNN training. Lobster distinguishes between the I/O load of each individual GPU at fine granularity and coordinates the I/O operations of the GPUs at the node level, flexibly allocating available I/O bandwidth and threads as needed to reduce I/O load imbalance.

Lobster also coordinates the data loading and preprocessing stages of the pipeline, flexibly allocating threads between them to reduce bottlenecks. This coordination is achieved through the use of performance modeling, which we combine with reuse distance theory to design efficient eviction policies for distributed caching of the training samples. Such an optimized eviction policy complements state-of-the-art distributed caching approaches based on prefetching by avoiding the undesirable effect of evicting training samples that are needed in the near future in order to make room for prefetched samples that are needed later. We show that this method increases the cache hit ratio by 14.3% compared with state-of-the-art prefetching approaches such as that used in NoPFS [22].

Besides the above contributions, Lobster introduce other performance optimization techniques, such as co-locating data loading and preprocessing in the same NUMA node considering the potential impact of NUMA on the training pipeline,

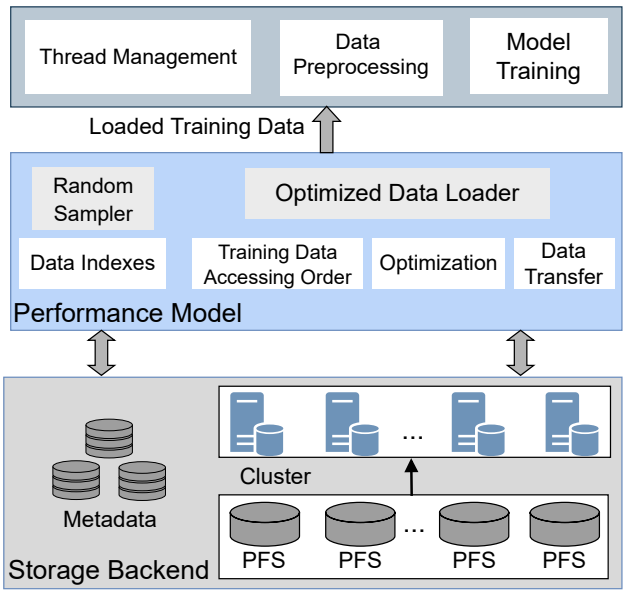


**Figure 5.1:** Overview of Lobster.

and concurrency throttling for data preprocessing to save working threads. Thanks to these contributions, Lobster is able to maximize GPU and cache utilization, thus hiding the overheads of data loading and enabling high performance and scalable end-to-end DNN training.

We characterize the performance (especially I/O performance) across 64 GPUs in a production environment for distributed DNN training, highlighting the I/O load imbalance across GPUs and frequent performance bottleneck shifts between data loading/pre-processing pipeline and the training process. This study reveals new opportunities for I/O performance optimization that are not considered by state-of-art approaches. We propose a thread management strategy to coordinate the resource usage between data loading and preprocessing in the training pipeline, as well as to mitigate the I/O load imbalance between GPUs. We introduce a holistic performance model that bridges the thread management strategy with a distributed caching proposal that features prefetching support and optimized eviction based on reuse distance. We design and implement a heuristic strategy to solve the optimization problem resulting from the performance model. This strategy consists of two phases (prefetching and eviction) and guides both the allocation of the threads and the distributed caching. We evaluate Lobster on a 64-GPU (NVIDIA A100) cluster and compare its performance with the state-of-the-art PyTorch I/O [82], DALI [83], and NoPFS [22] systems on several DNN models and training datasets.

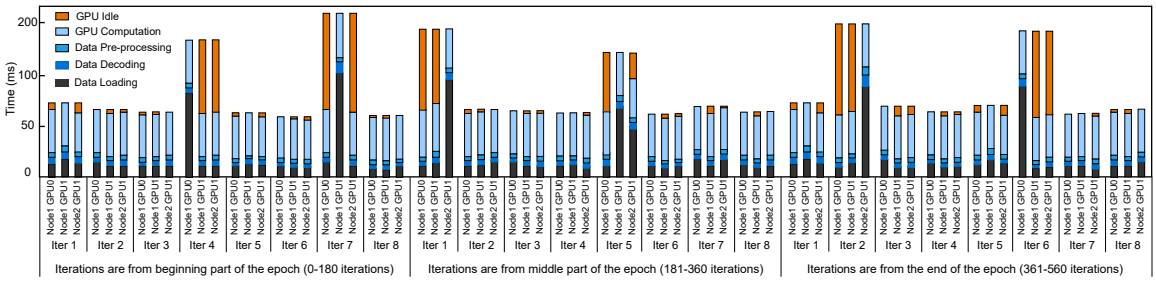


**Figure 5.2:** Execution time breakdown for the training pipeline on three GPUs, two on one node and the third on a second node.

## 5.2 Motivation

We now motivate our approach by examining the challenges introduced in Section 1 in detail. To this end, we run a series of experiments that profile the performance of the DNN training pipeline and discuss our observations.

A detailed description of the experimental setup is available in Evaluation section. Using this experimental setup, we train a ResNet50 model on the ImageNet-1K dataset using PyTorch 1.8 as the DNN runtime and DALI [83], an industry-standard state of art data loading library. We use a data-parallel setup deployed on eight nodes, for a total of 64 GPUs. Since the stages of the training pipeline are overlapping and we are interested in studying the bottlenecks, we measure the duration of the delays caused by each stage along the critical path.

Figure 5.2 shows detailed results for three GPUs: two co-located on the same node, and the third on a different node. We omit the first epoch (because the caches need to warm up and therefore the behavior is different compared to the rest of the epochs) and focus on 24 iterations (out of 562) of the second epoch: eight each in the beginning, middle, and end—enough to capture a recurring pattern throughout the epoch. We make the following observations:

**Observation 1: There is data load imbalance across GPUs.** GPUs are often idle during an iteration, but not because they are waiting for their own data loading and pre-processing stages, which are faster than, and therefore fully overlap with, training. The problem is rather that other GPUs have longer data loading and preprocessing stages, which causes them to become stragglers and delay the start of the training stage. As each GPU needs to perform the same amount of work during the training stage, the stragglers cause other GPUs to sit idle while they wait to

average the gradients during the backward pass. For example, during iteration 7, GPU0 of Node1 and GPU1 of Node2 are less loaded than GPU1 of Node1. Their GPU idle time takes 73% and 12% of the total iteration time, respectively. Compared with iteration 2, where there is no data load imbalance, iteration 7 is 3× slower. Our results show that such data load imbalances occur frequently: in 65.3% of our 562 iterations.

**Observation 2: Data loading overheads vary frequently and irregularly across iterations, leading to the performance bottleneck shifting among the stages of the pipeline.** Since the pipeline overlaps the stages, the slowest stage becomes a bottleneck. Ideally, data loading and preprocessing should never become a bottleneck. However, not only does this happens frequently with state-of-art approaches, but it is difficult to predict: during the same iteration, on some GPUs the training stage is the performance bottleneck, while the opposite is true on the other GPUs. Similarly, on the same GPU, the bottleneck can shift between data loading and training across iterations and exhibit an irregular pattern. This can happen on any GPU. In this experiment, we did not observe the preprocessing stage becoming a bottleneck by itself, however, this can happen and was reported by other studies [35]. When data loading is the bottleneck, training performance suffers significant slowdown. For example, in the case of GPU0 on Node0, in the two iterations where data loading is the bottleneck, its duration is 3× longer than the training stage—an observation that is explained by the fact that remote I/O to the external repository and/or the local caches of other compute nodes is orders of magnitude slower that local I/O. Furthermore, this effect is also correlated with load imbalance: whenever the performance bottleneck shifts in a GPU, other GPUs tend to exhibit load imbalance. Thus, when data loading is the bottleneck, it tends to exhibit a bursty pattern.

**Observation 3: The preprocessing stage does not benefit from an arbitrarily large number of threads.**

Data preprocessing is characterized by a streaming memory access pattern: training samples are continuously arriving in batches, and each training sample can be further split into sub-domains (e.g. regions in an image). The computations are typically embarrassingly parallel, therefore they can considered a bag of tasks to be assigned to threads. Varying the number of threads changes data parallelism and

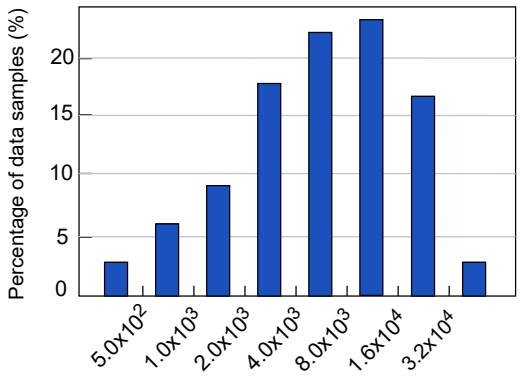


**Figure 5.3:** Histogram of the reuse distance of the training samples, measured in terms of numbers of iterations (X-axis)

memory bandwidth consumption, which in turn impacts performance. To study this effect, we vary the number of preprocessing threads and measure the preprocessing throughput (decoding/decompression and data augmentation). As can be observed in Figure 5.4, the preprocessing throughput peaks at 6 threads, after which it flattens and even slightly becomes worse. This effect has been observed by others as well [84, 85]. For data preprocessing, intensive memory bandwidth consumption is the major performance bottleneck when the number of threads is large. Furthermore, excessive memory bandwidth consumption also impacts the other stages in the pipeline. Therefore, it must be avoided.

**Observation 4: Many training samples have a long reuse distance.**

During data-parallel training, each GPU processes a different mini-batch. This means that each training sample cached on a compute node at iteration $i$ may be reused the first time either by the same or a different GPU co-located on the same node at iteration $j$. We call $j - i$ the *reuse distance*. Studying the reuse distance of the training samples is important to understand how well the cache of each compute node is utilized. Figure 5.3 shows the histogram of reuse distance of data samples accessed by GPUs for Node1. We observe that many training samples have a long reuse distance. Here, when two memory accesses to a sample are separated by at least one epoch away, we call it "long". For example, 80% of the training samples have the reuse distance larger than 1,000 iterations.

**Implications.** Observation 1 indicates that the data load imbalance is frequent and leads to stragglers. Thus we need a fine-grained load balancing strategy that is aware of individual GPUs. Observation 2 is correlated to Observation 1 and points

to a frequent shift of the performance bottleneck among the stages of the pipeline due to I/O bursts. Therefore, it is important to (1) optimize the utilization of the node-local cache to avoid remote I/O; (2) coordinate with the other stages of the pipeline and allocate more I/O threads to data loading when remote I/O cannot be avoided. Observation 3 indicates that the preprocessing stage does not benefit from an arbitrarily large number of threads and it can be even detrimental to allocate too many threads to it due to high memory bandwidth consumption. Therefore, it is important to determine the minimum number of threads needed to reach the peak preprocessing throughput and not exceed it. Observation 4 indicates that we can leverage long reuse distance of training samples to optimize the utilization of the node-local cache. In particular, we can explore new eviction policies that coordinate with prefetching to avoid the undesirable situation in which the training samples that are needed in the near future are evicted at the expense of the prefetched training samples that are needed later.

## 5.3 Design

Based on the observations summarized in Motivation section, we propose Lobster, a holistic data loading I/O runtime for distributed DNN training. Lobster uses the following high-level strategy to balance the work of the different stages of the training pipeline between different GPUs: (1) decide the number of data preprocessing threads; (2) given the number of data preprocessing threads, when the data loading is not a performance bottleneck of the training pipeline, decide the number of data loading threads per iteration for each GPU ; (3) given the number of data preprocessing threads, when the data loading is a performance bottleneck of the training pipeline, use performance modeling and run an heuristic algorithm to decide the number of data loading threads to balance I/O between GPUs in the same node. The heuristic algorithm considers the reuse distance of the training samples, which is used to coordinate with the prefetching and improve the overall hit ratio of the node-local cache. This strategy is implemented as illustrated in Overview section.

Lobster addresses (1) by throttling thread-level parallelism in preprocessing so that it can redirect threads for data loading. To decide the number of preprocessing threads, Lobster predicts the preprocessing throughput based on a piece-wise linear

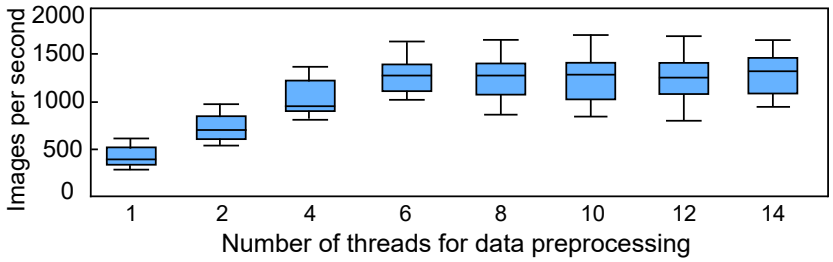


**Figure 5.4:** The impact of number of preprocessing threads (X-axis) on data preprocessing throughput (Y-axis)

regression model. For (2), Lobster introduces a multi-queue data structure to distinguish I/O between GPUs, and assigns threads to GPUs in proportion to data loading intensity. For (3), Lobster formulates the problem of deciding data loading threads as an optimization problem, and uses a computation-efficient heuristic algorithm to solve it. We now examine each of these aspects in detail.

### 5.3.1 Flexible Preprocessing Thread Management

We decide the number of data preprocessing threads based on two goals: (1) the combined duration of preprocessing and data loading should be smaller than the training stage; (2) need to redirect threads to improve the performance of the data loading stage.

Given a batch of training samples, we use the following two-step algorithm to meet the above goals. **Step 1:** predict the preprocessing throughput using the optimal number of preprocessing threads (that reaches the peak preprocessing throughput, as explained in Motivation section. The prediction is based on performance modeling, as detailed below. Then, we use the methods in Sections 5.3.2 and 5.3.3 to decide the number of threads allocated for the data loading stage. This aims to reach goal (1). **Step 2:** as long as goal (1) is not reached and the preprocessing stage is not a performance bottleneck, take away one thread from the preprocessing stage and make it available for data loading. The frequency of running this algorithm can be adjusted to reach a trade-off where we avoid excessive overheads on one hand, while maintaining the capability to adapt quickly to changing performance bottleneck shifts.

The success of the above algorithm depends on the accuracy of the performance

predictions of the data preprocessing stage. To this end, for a specific training sample size, we build a piece-wise linear regression model that takes the number of threads as input and predicts the execution time of processing one training sample. We build a portfolio of models, each of which corresponds to a training sample size. During runtime, if the sample size does not have a corresponding model in the portfolio, we choose the model whose sample size is closest to the one considered.

Note that the performance modeling approach mentioned above is architecture-dependent and data sample-dependent. This means that for different training environments (different hardware, types of preprocessing and sample sizes) we need to adjust the performance modeling. However, in practice, the same HPC machines, types of preprocessing and sample sizes are reused across many training instances of the same or different DNN models. Therefore, the cost of constructing performance model is amortized.

### 5.3.2 Coordinated Data Loading / Preprocessing

Given a fixed number of threads for the data preprocessing stage, the remaining CPU threads of the same node will be assigned for the data loading stage that serves all GPUs of that node.

**Discriminating data load overheads between co-located GPUs.** Current state of art efforts [82, 83, 36, 35, 22] serve all GPUs equally using a pool of threads reserved for the data loading stage. However, this is sub-optimal because the GPUs that trigger higher data loading overheads should be served using more threads, such that they will not become stragglers. To address this issue, Lobster proposes to maintain a separate request queue for each GPU, each of which can be assigned a different number of threads such as to achieve load balancing. Note that the data loading requests are placed in the queue of each GPU based on deterministic prefetching while considering the reuse distance. These details will be discussed later.

**Thread assignment.** Given the requests in all GPU queues of a node that correspond to future training iterations, Lobster checks if there is a GPU that is predicted to become a straggler due to data loading. The prediction is based on performance modeling, as detailed in our proposed performance model. Assignments are then made by using the heuristic detailed in our proposed heuristics section. When a GPU

**Table 5.1:** Notation used in performance models.

| Notation | Metric | Description |
|---|---|---|
| $N$ | | number of compute nodes |
| $M$ | | number of GPUs in one compute node |
| $Mem$ | | storage space on each compute node |
| $D$ | | training dataset, comprising $\|D\|$ data samples |
| $S$ | | size of $D$ |
| $s_i$ | | size of data sample $d_i$ ($d_i \in D$) |
| $I$ | | number of iterations per epoch |
| $T_l(\alpha)$ | $MB/s$ | local memory read throughput ($\alpha$ read threads) |
| $T_r(\beta)$ | $MB/s$ | inter-node read throughput ($\beta$ read threads) |
| $T_{PFS}(\gamma)$ | $MB/s$ | read throughput of remote PFS ($\gamma$ read threads) |

is not predicted to become a straggler, the number of threads assigned to the request queue is proportional to the size of the queue.

### 5.3.3 Performance Model

We introduce a holistic performance model that bridges the thread management strategy for data loading and preprocessing (discussed above) with distributed caching. Table 6.1 summarizes the major notations used by our model.

Let $s_i$ be the size of sample $d_i$ in the training dataset $D$, which thus has a total size of $S = \sum_{0 \le i < |D|} s_i$; $N$ be the number of nodes and $M$ the number of GPUs per node (for a total of $N \times M$ GPUs); $Mem$ be the host memory size allocated for caching; and $|B|$ be the mini-batch size. If $S > Mem$, then the training data cannot be fully cached on a single node; if $S > N \times Mem$, it cannot be fully cached across all nodes. One epoch consists of $I = \left\lceil \frac{|D|}{|B| \times M} \right\rceil$ iterations, or $I = \left\lfloor \frac{|D|}{|B| \times M} \right\rfloor$ iterations if we discard the last (potentially partial) iteration. At iteration $h$ ($0 \le h < I$), each GPU $G_i^j$ (where $i$ is the node ID and $j$ is the GPU ID), processes its own mini-batch $B^{h,i,j}$.

Overall, the GPUs need to read a collection $B^h = \bigcup_{i \in N, j \in M} B^{h,i,j}$ of training samples concurrently. Three scenarios can arise when reading training sample $d_k$ with size $s_k$ on node $n_i$:

1. $d_k$ is present in the local cache of node $n_i$, in which case the data loading duration is $\frac{s_k}{T_l(\alpha)}$, where $T_l(\alpha)$ is the local cache read throughput with $\alpha$ concurrent I/O threads.

2. $d_k$ is present in the remote cache of another node, in which case the data loading

duration is $\frac{s_k}{T_r(\beta)}$, where $T_r(\beta)$ is the remote cache read throughput of a single I/O with $\beta$ concurrent threads.

3. $d_k$ is present on the remote storage repository (parallel file system), in which case the data loading duration is $\frac{s_k}{T_{PFS}(\gamma)}$, where $T_{PFS}(\gamma)$ is the PFS read throughput of a single I/O thread with $\gamma$ concurrent I/O threads (for simplicity, we assume $T_{PFS}(\gamma)$ to be globally stable on the average across the compute nodes).

Assume $B_{HL}^{h,i,j}$ and $B_{HR}^{h,i,j}$ represent the training samples that cause cache hits on node $n_i$'s local cache and on the cache of remote nodes respectively, while $B_M^{h,i,j}$ represents training samples that cause cache misses and need to be fetched from the PFS. We have $B^{h,i,j} = B_{HL}^{h,i,j} \cup B_{HR}^{h,i,j} \cup B_M^{h,i,j}$. Assuming $T_L(n_i, B^{h,i,j})$ represents the duration of loading $B^{h,i,j}$ for GPU $G_i^j$, we have:

$$T_L(n_i, B^{h,i,j}) = \frac{\sum_{d_k \in B_{HL}^{h,i,j}} s_k}{\alpha_{i,j} \times T_l(\alpha_{i,j})} + \frac{\sum_{d_k \in B_{HR}^{h,i,j}} s_k}{\beta_{i,j} \times T_r(\beta_{i,j})} + \frac{\sum_{d_k \in B_M^{h,i,j}} s_k}{\gamma_{i,j} \times T_{PFS}(\gamma i, j)} \tag{5.1}$$

In Equation 5.1, $\alpha_{i,j}$, $\beta_{i,j}$ and $\gamma_{i,j}$ are the initial number of data loading threads allocated for each scenario for each GPU $G_i^j$ (as discussed in subsection 5.3.2). Given the mini-batch $B^{h,i,j}$, we denote its data preprocessing time $T_P(n_i, B^{h,i,j})$. We assume the duration of the training stage $T_{train}$ is constant. Thus, in order to minimize the performance bottleneck introduced by the data loading and preprocessing stages for GPU $G_i^j$ during iteration $h$, we need to minimize the following expression:

$$\min \left| T_L(n_i, B^{h,i,j}) + T_P(n_i, B^{h,i,j}) - T_{train} \right| \tag{5.2}$$

However, our overall goal is to minimize the performance bottleneck introduced by the data loading and preprocessing of all $M$ GPUs of the compute node during iteration $h$. Assuming $T_{max}^{h,i}$ and $T_{min}^{h,i}$ are the maximum and minimum $T^{h,i,j}$ at iteration $h$ across all $M$ GPUs (where $T^{h,i,j}$ is the execution time of the $h^{th}$ iteration for the $j^{th}$ GPU on the node $i$), we can achieve this goal by minimizing the gap between $T_{max}^{h,i}$ and $T_{min}^{h,i}$ as follows:

$$\min \left| T_{max}^{h,i} - T_{min}^{h,i} \right| \tag{5.3}$$

Unfortunately, the solution for Equations 5.2 and 5.3 is an optimization problem that can be solved using Integer Linear Programming (ILP), which is known to be NP-complete [86]. Even if this were feasible for a single iteration $h$, we have to consider that we have a total of $N \times M$ GPUs and a large number $I$ of iterations. Thus, an exact solution to this optimization problem is not tractable.

### 5.3.4 Heuristic Strategy

To make the optimization problem introduced above tractable, we propose a heuristic strategy that works in two phases: (1) determine the number of data loading threads for each GPU; (2) determine an efficient eviction strategy for deterministic prefetching to avoid cache misses due to evicting samples with small reuse distance. Note that (2) influences the scenarios applicable for the training samples (node-local cache vs. remote cache vs. PFS), therefore there is a close connection between (1) and (2).

**Thread assignment in case of predicted stragglers.** Given a set of mini-batches to be prefetched by the GPUs co-located on a node, Lobster determines a near-optimal assignment of data loading threads by using a greedy algorithm that aims to satisfy the goals formulated in Equations 5.2 and 5.3.

The initial allocation $L_{th}$ of data loading threads to each co-located GPU is proportional to the number of pending requests in the data loading queue. We then calculate the difference between the duration of data loading + preprocessing and that of the training stage by using Equations 5.1 and 5.2. If this difference is greater than a threshold $\tau$ (which can be fine-tuned as needed to prune the search space), then we employ a binary search to explore the search space until we converge to a near-optimal solution that minimizes the gap between $T_{max}$ and $T_{min}$.

To cover the case when the greedy algorithm does not converge, we introduce an array $\mathcal{W}$ whose length is $T_L$, the maximum number of data loading threads that can be used by a node. The array records $T_{dif}$ calculated in the prior iterations. When the array is fully populated, then we stop the search and choose the solution that has the minimum $T_{dif}$ among all those recorded in $\mathcal{W}$.

**Eviction Policy based on Reuse Distance and Deterministic Prefetching** It is important to note that the determinism of the prefetching pattern of one

node is a *global* property: it is known to *all* other nodes (e.g. by fixing the pseudorandom number generator seed of each node such that it is a function of a fixed seed and the node id). Thus, we can determine, at each moment during training, two parameters: (1) how many times each training sample will be reused by all GPUs until the end of training; (2) the minimum reuse distance of each training sample across all GPUs. To obtain these parameters efficiently, we maintain a list of future accesses for each training sample. Each entry in the list records the GPU and iteration number during which the training sample needs to be accessed for the reminder of the training. Based on this list, we apply two sub-policies:

**Reuse count policy.** If during training the number of accesses to a training sample reaches the reuse count for a node, then the sample is evicted from the node-local cache—unless no other node in the group holds a copy, as eviction would then force the other nodes to perform expensive I/O operations to re-prefetch the training sample from the remote storage repository.

**Reuse distance policy.** Let $I$ be the number of iterations in an epoch, $h$ the current iteration, and $B^h$ the set of mini-batches accessed by all co-located GPUs on a node. Then after iteration $h$ has finished, we can check the next reuse distance of each training sample $d_k \in B^h$. If the next reuse distance is larger than $2 \times I - h$, then the training sample will not be accessed by any GPUs on the node during the next epoch. In this case, the training sample can be considered as being reused far enough in the future to justify eviction in order to make room for more prefetches.

**Coordination with prefetching.** Thanks to the two eviction policies mentioned above, any spare capacity in the node-local cache can be used for prefetching more training samples. However, if the training samples can be prefetched faster than they are consumed, the spare capacity will be quickly filled. In this case, we can evict the training samples with the largest reuse distance, while prioritizing the prefetches with the nearest reuse distance.

### 5.3.5 Implementation Details

Lobster consists of two components: one is used in offline fashion to construct piece-wise linear regression models for the preprocessing stage and to pre-compute an efficient thread management plan combined with an efficient prefetching/eviction

plan based on the reuse distance. The planning phase is based on a simulator proposed by [22], which was extended to: (1) decide the number of data preprocessing threads; (2) decide the number of data loading threads; (3) adding the cache eviction algorithm, and (4) add coordination logic between data preprocessing and data loading.

The other component is an online runtime implemented in C++ and built on the top of DALI 2.0 [83]. It is designed to interpret the plan generated by the offline component, and to enforce the thread management and data prefetching as planned.

## 5.4 Evaluation

We evaluate the performance of Lobster in two data-parallel scenarios: (1) single node with multiple GPUs and (2) multiple nodes, each with multiple GPUs. In each case, we compare Lobster with three baseline approaches in terms of I/O performance, memory cache efficiency, and end-to-end training runtime. Our evaluation aims to answer four questions:

- Does Lobster have better I/O performance than the baselines?
- Does Lobster address the load imbalance problem?
- Does Lobster influence end-to-end training performance compared with baselines?
- Does each component of Lobster contribute to the overall improvement?

### 5.4.1 Experimental Setup

We evaluate Lobster with six representative DNN models that are frequently used as training benchmarks (ResNet50 [87], ResNet32 [87], ShuffleNet [88], AlexNet [89], SquenceNet [90], VGG11 [91]) and, for each model, two datasets, IMAGENet-1K and ImageNet-22K. We use PyTorch 1.8 with NCCL2 for all evaluations. At the beginning of DNN model training, the training datasets are stored on a Lustre parallel file system mount point.

**Baselines.** We compare Lobster with three baseline approaches:

- PyTorch I/O [82]: The built-in PyTorch DataLoader using a constant number of threads for data loading and another constant number of threads for preprocessing.
- DALI [83]: A widely used NVIDIA library for DNN training I/O. DALI uses three threads for data loading by default and leaves other threads for preprocessing.
- NoPFS [22]: A state-of-the-art approach that implements deterministic prefetching that is combined with PyTorch. The thread management for NoPFS is the same as that with PyTorch I/O.

**Hardware.** We performed experiments on Argonne's ThetaGPU HPC machine, which is specifically optimized for training DNNs at scale. It comprises 24 NVIDIA DGX A100 nodes, each of which is equipped with eight NVIDIA A100 Tensor-Core GPUs, two AMD Rome CPUs, 1 TB of DDR4 memory and 320 GB GPU memory. This amounts to a total of 24 TB DDR4 and 7.6 TB GPU memory. We use 40 GB DDR4 memory as node-local cache on each node. If the cache is large, all samples are placed locally without causing I/O. But typically, a small portion of DDR4 memory is used as cache. The nodes are interconnected with 20 Mellanox QM9700 HDR200 40-port switches wired in a fat-tree topology. The external storage provided by a Lustre parallel file system deployment provides an aggregate 250 GB/s bandwidth, mounted using POSIX.

**Datasets.** We use two widely used datasets with different sizes.

- ImageNet-1K [92]: This dataset, widely used to evaluate DNNs for image classification, consists of 1.28 million training images and 50,000 validation images, each image assigned to one of 1000 classes. Its total size is 135 GB.
- ImageNet-22K [93]: A representative larger dataset widely used to pre-train DNNs. This dataset consists of 14,197,103 training images and 7000 validation images, most with an image size of between 10 KB and 50 KB, and each assigned to one of 21,841 classes. The total dataset size is 1.3 TB.

### 5.4.2 I/O Performance

**Single-Node Multi-GPU Data-Parallel Training.** We run Lobster on a single node with eight GPUs. Figure 5.5(a) and (b) show the results for ImageNet-1K and

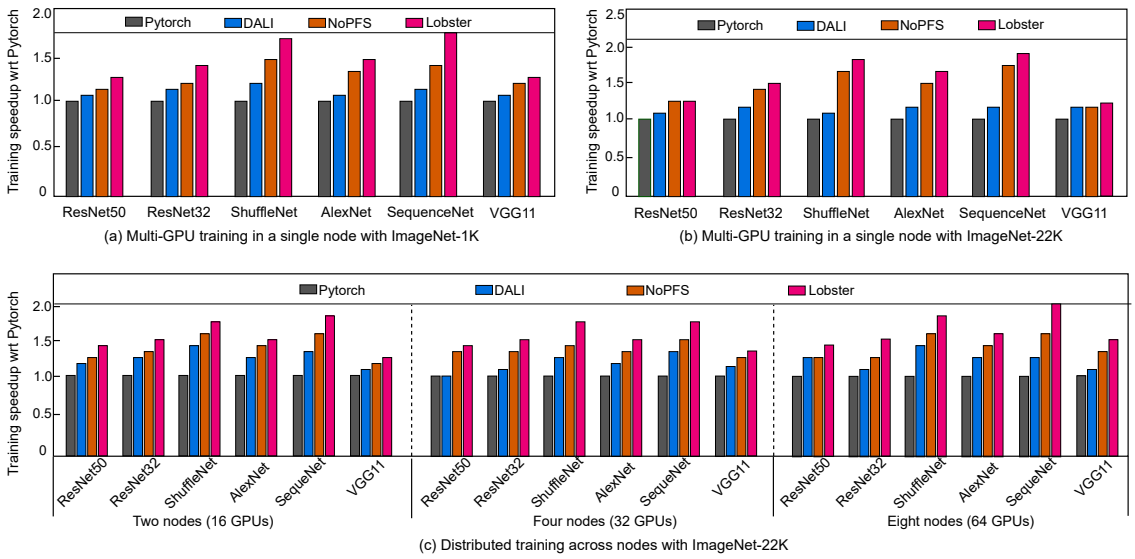


**Figure 5.5:** Comparison between Lobster and the baselines for multi-GPU training on a single node and distributed training across multiple nodes.

ImageNet-22K, respectively. We observe that:

(a) Lobster is 1.6× and 1.8× faster than PyTorch DataLoader using ImageNet-1K and ImageNet-22K, respectively. This performance improvement results mainly from the alleviation of I/O load imbalance between GPUs. The performance improvement is especially large in the case of ImageNet-22K (the larger dataset).

(b) Lobster is 1.7× faster than DALI, for two reasons: (1) DALI lacks fine-grained thread-level optimizations for the training pipeline, while Lobster can flexibly allocate CPU threads to coordinate data loading and pre-processing; and (2) Lobster is NUMA-aware, and co-locates data loading and preprocessing threads.

(c) Lobster is 1.2× faster than NoPFS. This is due to the cache eviction based on reuse distance, which complements deterministic prefetching. Specifically, NoPFS evicts the training samples to accommodate the training samples to be prefetched for the next iteration, while our approach is able to prefetch more training samples thanks to its eviction policies, which translates to higher cache hit rates and better performance.

**Multi-Node Distributed Data-Parallel Training.** We evaluate Lobster on eight nodes when using all eight GPUs available on each node. We see in Figure 5.5(c) that for ImageNet-22K Lobster is 2.0×, 1.4×, and 1.2× faster than PyTorch DataLoader, DALI, and NoPFS, respectively. Compared with the single-node evaluation, Lobster

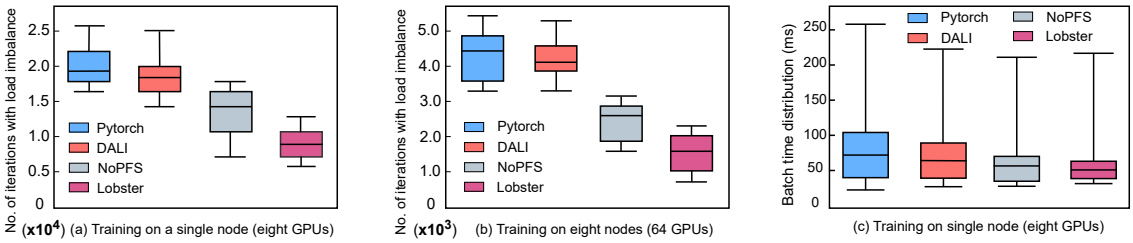


**Figure 5.6:** The number of iterations with load imbalance and the distribution of batch time. We use ResNet50 with ImageNet-1K.

makes better use of the distributed cache across nodes, leading to a larger performance improvement.

**Scalability of Data-Parallel Training.** We evaluate Lobster using a variable number of nodes and with different datasets. For ImageNet-1K, we use a single node whose memory cache is smaller than the dataset size, while for ImageNet-22K, we use multiple nodes whose aggregated memory cache across the nodes is smaller than the dataset size (but larger than the size of ImageNet-1K). Figure 5.5 shows the results with respect to PyTorch Dataloader: (a) Lobster scales well for ImageNet-1K. Furthermore, when training with ImageNet-22K, compared with PyTorch Dataloader, Lobster has a speedup of 1.53× on average (up to 1.9×); (b) with different system scales on a single node and multiple nodes, Lobster consistently shows a significant speedup (1.2×–2.0×).

### 5.4.3 Reduction of Load Imbalance

To evaluate Lobster's effectiveness in reducing data load imbalance, we count the number of iterations with load imbalance across GPUs in each epoch. We use ResNet50 with ImageNet-22K as the training dataset. Figure 5.6 depicts the results.

**Single-Node Multi-GPU Data-Parallel Training.** We train the model for 50 epochs, each of 55,457 iterations. We depict in Figure 5.6(a) the results for all epochs. Compared with PyTorch, DALI, and NoPFS, Lobster reduces the iterations with load imbalance by 31.4%, 16.4%, and 7.9% respectively on average. WithLobster, only 17.5% of all training iterations exhibit load imbalance.

**Multi-Node Distributed Data-Parallel Training.** We fix the number of iterations per epoch at 6932 and train ResNet-50 for 50 epochs. Figure 5.6(b) shows the results for all epochs. The number of iterations is smaller than when training

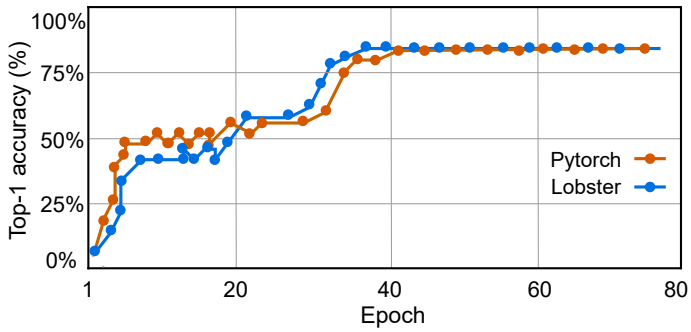


**Figure 5.7:** Training accuracy curve for training ResNet50 on ImageNet-1K using eight nodes (64 GPUs) with default ResNet50 hyperparameter settings.

on a single node because we use more GPUs. Compared with PyTorch DataLoader, DALI, and NoPFS, Lobster reduces the number of iterations with load imbalance by 35.2%, 25.8%, and 9.7% respectively. Overall, with Lobster, only 22.8% of all training iterations still exhibit load imbalance.

Lobster is able to reduce the number of iterations with load imbalance more effectively thanks to the coordination between the data loading and preprocessing stages, which redirects more threads for data loading when the GPUs are bottlenecked by it.

**Batch time distribution.** Figure 5.6(c) presents the batch time distribution when training ResNet50 with ImageNet-1K on one node (8 GPUs), showing successful mitigation of performance degradation brought by the load imbalance across GPUs. With Lobster, there is less variance in per-batch time (iteration duration) than the baselines. Lobster's batch time is also shorter than other methods'. Figures 5.6 demonstrates a key performance advantage of Lobster: reducing the batch time where the data loading is slow due to load imbalance across GPUs.

### 5.4.4 End-to-End Training

Lobster does not change the randomness of data accessing during the distributed training. The techniques in Lobster do not influence the DNN model's accuracy. To demonstrate this, we train ResNet50 to convergence for ImageNet-1K on eight nodes (64 GPUs) with both Pytorch DataLoader and Lobster. Figure 5.7 shows the accuracy curves. We see that the two methods have similar learning curves, although with some slight variation due to different random seeds for network parameters. In both cases,

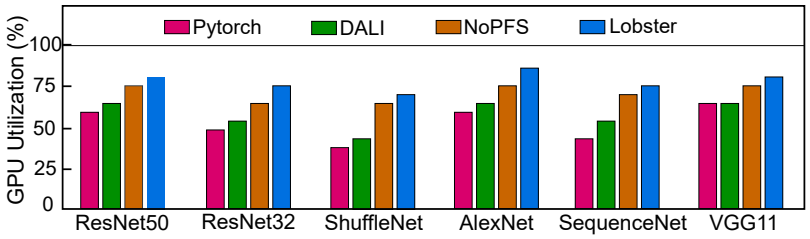


**Figure 5.8:** GPU utilization when training ResNet50 on ImageNet-1K using one node (eight GPUs). X-axis represents different DNNs used for testing and Y-axis is the GPU utilization.

training converges to the target accuracy of 76.0% in around 40 epochs. Nevertheless, as Figure 5.5 indicates training with Lobster is up to 1.4× faster than with PyTorch DataLoader. As a consequence, using Lobster for data loading achieves an overall shorter training time.

### 5.4.5 Resource Utilization

**Memory cache hit ratio.** We measure the cache hit ratio of the memory cache during the whole training process. We use one node with eight GPUs and ImageNet-1K. Lobster has higher cache hit ratio than the baselines. On average, the cache hit ratio with Lobster is 63.2%, while it is 24.5%, 32.6%, and 48.9% with PyTorch DataLoader, DALI, and NoPFS respectively. The higher cache hit ratio demonstrates the effectiveness of cache eviction as a complement for deterministic prefetching. NoPFS has higher cache hit ratio than PyTorch DataLoader and DALI, because of its efficient distributed cache with deterministic prefetching. However, its cache hit ratio is lower than Lobster because of a simpler cache eviction policy.

**GPU utilization.** We measure average GPU utilization during the whole training process. We use one node with eight GPUs and ImageNet-1K, and fix the number of epochs at 50. As can be observed in Figure 5.8, Lobster has higher GPU utilization than the baselines: 76.1% vs. 52.3%, 57.5%, and 72.4% for PyTorch DataLoader, DALI, and NoPFS respectively. The higher GPU utilization of Lobster demonstrates the effectiveness of addressing load imbalance across GPUs.

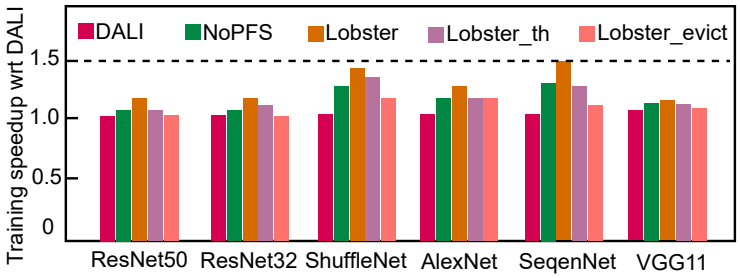


**Figure 5.9:** Ablation study of Lobster when training ResNet50 on ImageNet-1K using one node (eight GPUs). Y-axis is the training time speedup compared with DALI.

### 5.4.6 Ablation Study

We next evaluate the individual impacts of Lobster's thread management and its cache eviction policies. To this end, *Lobster_th* includes thread management but excludes cache eviction based on reuse distance, while *Lobster_evict* does the precise opposite. We show in Figure 5.9 results for ImageNet-1K on a single node (eight GPUs).

**Thread management**. We see that: (1) the thread management optimization contributes more to the training performance improvement than the cache eviction policy; (2) the thread management improves the training performance by up to 1.4× (1.3× on average), compared with DALI.

**Cache eviction policy**. We also see that the cache eviction policy based on reuse distance (1) leads to 15% higher performance than DALI, on average, and (2) is more helpful for small models (e.g., ShuffleNet, SequenceNet), for which the duration of the training stage is smaller compared with the larger models (and thereby less likely to become a performance bottleneck).

## 5.5 Summary

Data loading is becoming a major performance bottleneck in distributed DNN training. Prior studies of data loading performance for distributed DNN training have conducted neither a holistic analysis of all training pipeline stages not a fine-grained analysis of the load of individual GPUs, two areas that present opportunities for further optimization. To fill this gap, we have proposed Lobster, a data loading runtime that exploits several observations related to load imbalance, performance

bottlenecks in various stages of the training pipeline, and the reuse distance of training samples to propose a new flexible thread management strategy and cache eviction policy that complements deterministic prefetching. These methods allow Lobster to consistently outperform the state-of-art PyTorch I/O, DALI, and NoPFS systems by 1.3–2.0×.

# Chapter 6

# ArbiLIKE: An Accurate Cardinality Estimator for Arbitrary LIKE Predicates

## 6.1 Overview of ArbiLIKE

Figure 6.1 provides a high-level overview of ArbiLIKE's architecture, which consists of four stages. In the *statistical information collection* stage, ArbiLIKE collects valuable statistics offline by parsing string tuples from DB, including a pair (substring, cardinality) built using the $q$-grams technique and an inverted list (§6.2) for each substring.

In the *predicates encoding* stage (§6.3), ArbiLIKE begins by using a bottom-up hierarchical clustering technique (§6.3.2) to organize the offline-gathered pairs of (substring, cardinality). This clustering relies on the cardinality-distances between different substrings. Such a method ensures that within a cluster, pairs of (substring, cardinality) exhibit minimal cardinality variations, while pairs in distinct clusters show substantial cardinality differences. Following this, ArbiLIKE adopts a multi-tiered contrastive embedding strategy to transform substrings into feature vectors (§6.3.4), using the formed hierarchical clusters as the input.

In the *model design* stage (§6.4), ArbiLIKE introduces a novel sequence model, $\mathcal{M}_{est}$, designed to capture the importance of various substrings (§6.4.2). Upon completion of training, the model is ready to estimate cardinalities for LIKE predicates.

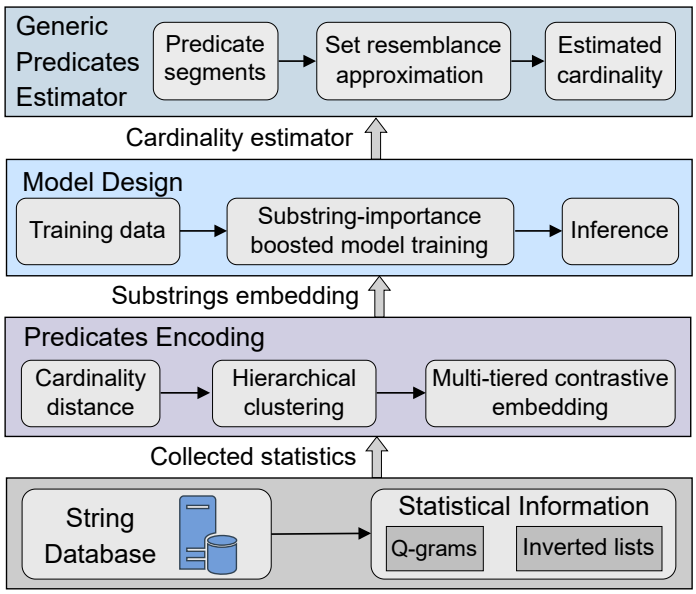


**Figure 6.1:** ArbiLIKE overview.

For each input LIKE predicate, ArbiLIKE transfroms it into a feature vector, which is subsequently fed into $\mathcal{M}_{est}$. The output of $\mathcal{M}_{est}$ represents the estimated cardinality for the corresponding predicate.

In the *generic predicate estimation* stage (§6.5), ArbiLIKE is adapted to address arbitrary LIKE predicates with any number of wildcards ("%", "_"). Initially, a generic LIKE predicate is decomposed into one or multiple LIKE sub-predicates based on wildcard positions. Then, using $\mathcal{M}_{est}$ (§6.4.2), ArbiLIKE estimates the cardinality for each sub-predicate and compute a resemblance value (§6.5.2) among them. The resemblance value and the cardinality estimates of the sub-predicates determines the cardinality of a generic LIKE predicate.

## 6.2 Problem Description

In this section, we introduce some notations and preliminaries. The definitions of the notations are summarized in Table 6.1.

**Notations.** Consider a string database DB, containing $N_S$ string tuples. We represent the database as $\mathsf{DB} = \{S_i\}_{i=1}^{N_S}$, where $S_i$ is the $i$-th string tuple in DB ($1 \leq i \leq N_S$). Each string tuple $S_i$ in DB is composed of a sequence of characters. For a given string $S_i$, we denote $S_i[j:k]$ as the substring that starts at the $j$-th position

**Table 6.1:** Notations.

| Notation | Description |
|---|---|
| $\mathsf{DB} = \{S_i\}_{i=1}^{N_S}$ | a database consisted of $N_S$ string tuples |
| $S_i$ | the $i$-th string tuple in $DB$, $i \in [1, N_S]$ |
| $S[j:k]$ | the substring of $S$ from position $j$ to position $k$ |
| $\ell_i$ | length of string tuple $S_i$ |
| $\Sigma$ | the alphabet of string tuples in $\mathsf{DB}$ |
| $q_{ij}$ | the $j$-th $q$-gram of the string $S_i$ |
| $Q(S_i)$ | a set of $q$-grams of string tuple $S_i$ |
| $I(q_{ij})$ | the inverted list of $q$-gram $q_{ij}$ |
| $Sig(q_{ij})$ | the signature vector of $q$-gram $q_{ij}$ |

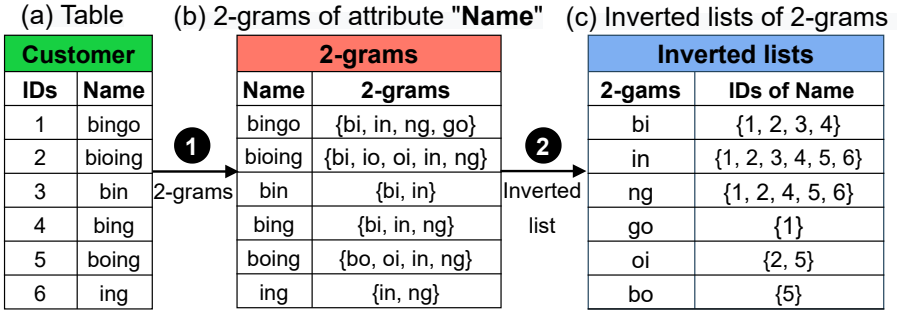


**Figure 6.2:** $q$-grams and inverted lists. (a) The Customer table; (b) All the $q$-grams for the "Name" attribute ($q = 2$); (c) Each row consists of a 2-gram and corresponding inverted list.

and ends at the $k$-th position of $S_i$, where $1 \leq j \leq k \leq \ell_i$. **LIKE predicates.**

The SQL supports two wildcards, "%" and "_", for specifying string patterns in LIKE predicates. The wildcard "%" allows for the substitution of zero or more characters in a string, while the wildcard "_" substitutes only one character in a string. A couple of examples are as follows: A query LIKE "%ab%" matches all string tuples in DB containing the substring "ab"; the query LIKE "a_b" matches all string tuples composed of three characters, with the first character being 'a' and the last character being 'b'. The query LIKE "%ab%cd%" selects all the string tuples containing "ab" followed by "cd", with any number of characters in between. A diverse range of LIKE predicates can be formulated using the two wildcard characters ("%", "_").

**Q-grams model.** For a given string tuple $S_i$ in DB, its substrings can be acquired via the $q$-grams technique [94]. We obtain $q$-grams of $S_i$ by sliding a window of size $q$ over the continuous characters of $S_i$, and each $q$-gram is a substring of $S_i$. We represent the set of $q$-grams of $S_i$ as $Q(S_i)$, written as $Q(S_i) = \{sub_{ij}\}_{j=1}^{|S_i|+2q}$,

where $sub_{ij}$ is the $j$-th $q$-gram of the string $S_i$, $|S_i|$ denotes the length of $S_i$, $Q(S_i)$ contains $|S_i| + 2q$ substrings. Figure 6.2(b) presents an example of $q$-grams derived from "Name" attribute in the Customer table (depicted in Figure 6.2(a)), and we set $q$ to 2 in this example.

**Inverted list for a substring in $Q(S_i)$.** Given $Q(S_i)$ of the string tuple $S_i$ in DB, the inverted list of a substring $sub_{ij}$ in $Q(S_i)$ is denoted as $\mathbf{I}(sub_{ij})$. $\mathbf{I}(sub_{ij})$ represents a set of string tuple IDs, defined in Equation 6.1.

$$\mathbf{I}(sub_{ij}) = \{i : S_i \in \mathsf{DB} \wedge sub_{ij} \in Q(S_i)\}_{i=1}^{N_S} \tag{6.1}$$

Equation 6.1 indicates that if a string tuple $S_i \in \mathsf{DB}$ contains the substring $sub_{ij}$, the string tuple's unique string ID is added into $\mathbf{I}(sub_{ij})$. The inverted list of a substring is leveraged to estimate the cardinality of arbitrary LIKE predicates in ArbiLIKE. Figure 6.2(c) shows an example of the inverted lists for the substrings (2-grams) of the "Name" attribute in the Customer table.

## 6.3 LIKE Predicates Encoding

We first discuss how to collect the statistics and encode Like predicates in this section.

### 6.3.1 Statistics Collection

Within a LIKE predicate, a string can be decomposed into one or multiple substrings. Our embedding methodology (§6.3.2 and §6.3.3) leverages both these substrings and their associated actual cardinalities in the database. **Collection of pairs of (substring, cardinality).** We employ the set $Sub(DB)$ to store pairs of (substring, cardinality) derived from string tuples in DB. To construct $Sub(DB)$, we traverse each string tuple $S_i$ in DB ($1 \leq i \leq N_S$) and extract all pairs of (substring, cardinality) using the $q$-grams technique. In ArbiLIKE, $q$ is set to 3 (see Table 6.5). Thus, $Sub(DB)$ comprises $\{(sub_i : card_i)\}_{i=1}^{N_{sub}}$, with $(sub_i : card_i)$ representing the $i$-th pair in $Sub(DB)$, and $N_{sub}$ is the count of those pairs.

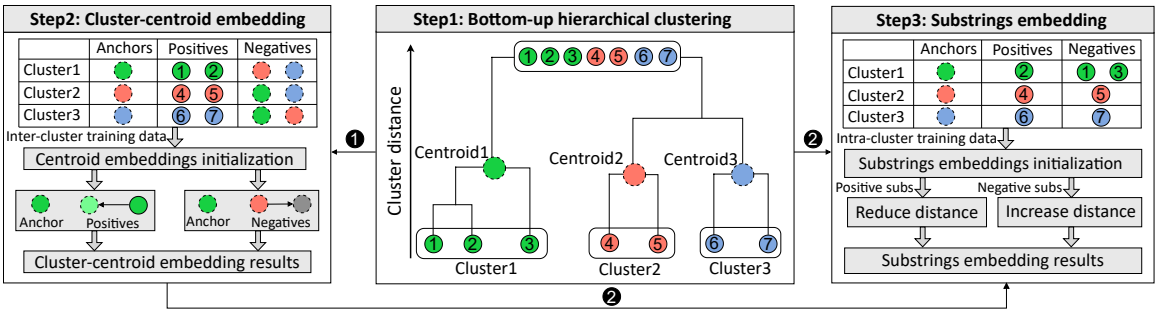


**Figure 6.3:** The cardinality-aware substring embeddings involves three steps. Each "node" is a pair of (substring, cardinality).

### 6.3.2 Cardinality-Distance Oriented Clustering

The predicate encoding methodology in ArbiLIKE is a two-phase process. First, it establishes clusters based on cardinality-distance using (substring, cardinality) pairs from $Sub(DB)$ (§6.3.1). Second, it harnesses a contrastive embedding technique (§6.3.3 and §6.3.4) to encode substrings from $Sub(DB)$ into feature vectors.

**Determine cardinality-distance.** In ArbiLIKE, we use the Q-error [95] and the edit distance [96] together to decide the cardinality-distance between two two unique (substring, cardinality) pairs, $P_i = (sub_i : card_i)$ and $P_j = (sub_j : card_j)$, within $Sub(DB)$. (1) For the Q-error, it offers a symmetric, scale-independent measure of the cardinality discrepancies and is prevalent in existing literature [95]. For $P_i$ and $P_j$, the Q-error of the cardinalities $card_i$ and $card_j$ is denoted by: $\text{Q-error}(\mathsf{card_i}, \mathsf{card_j}) = \frac{\max(\mathsf{card_i},\mathsf{card_j})}{\min(\mathsf{card_i},\mathsf{card_j})}$, where $\mathsf{card_i}$ and $\mathsf{card_j}$ represent the actual cardinalities of substrings $\mathsf{sub_i}$ and $\mathsf{sub_j}$, respectively. (2) For the edit distance, it represents the minimum number of edits (insertions, deletions, substitutions) needed to change one string into another. We didn't choose hamming distance [97], as it counts the number of positions at which the corresponding characters in two strings of equal length. For $P_i = (sub_i : card_i)$ and $P_j = (sub_j : card_j)$, the edit distance between $sub_i$ and $sub_j$ is denoted by: $\mathsf{Edit}(\mathsf{sub_i}, \mathsf{sub_j})$. A large edit distance implies a greater distance between $sub_i$ and $sub_j$, indicating lower similarity; whereas a small edit distance means a shorter distance between them, signifying higher similarity. Based on the Q-error and the edit distance, we calculate the cardinality-distance beween $P_i$ and $P_j$ as follows, $\mathsf{Dist}(\mathsf{P_i}, \mathsf{P_j}) = \frac{1}{\mathsf{Edit}(\mathsf{sub_i},\mathsf{sub_j})} \times \frac{\max(\mathsf{card_i},\mathsf{card_j})}{\min(\mathsf{card_i},\mathsf{card_j})}$.

**Build cardinality-distance oriented clusters.** We adadt a bottom-up hierar-

chical clustering method [98] to establish clusters from the (substring, cardinality) pairs in $Sub(DB)$, guided by the cardinality-distance. The tree-structured of this method supplies fine-grained cardinality differences among various clusters, thereby enhancing the embedding quality for LIKE predicates.

The procedure for constructing the hierarchical clusters involves the following steps. **Step1:** Initialization. Each pair $P_i = (\mathsf{sub_i} : \mathsf{card_i})$ in $Sub(DB)$ is initially treated as an individual cluster. **Step2:** Compute pairwise distances. Compute the distance between each pair of clusters based on the cardinality-distance metric. **Step3:** Determine clusters to merge. Identify and merge the two clusters that are close to each other. In ArbiLIKE, we adopt Ward's minimum variance [99] method to decide which two clusters to merge. This method seeks the merger that results in the smallest increase in the total within-cluster variance. For two clusters $\mathsf{C_i}$ and $\mathsf{C_j}$, and their merged cluster $\mathsf{C_{ij}}$, with $|\mathsf{C_i}|$, $|\mathsf{C_j}|$, and $|\mathsf{C_{ij}}|$ denoting the number of (substring, cardinality) pairs in $\mathsf{C_i}$, $\mathsf{C_j}$, and $\mathsf{C_{ij}}$ respectively, the increased squared error by merging $\mathsf{C_i}$ and $\mathsf{C_j}$ is calculated as below.

$$\Delta\mathsf{SSE}(\mathsf{C_i}, \mathsf{C_j}) = \frac{|C_i| \times |C_j|}{|C_i| + |C_j|} \times \mathsf{Dist}^2(\mathsf{P_{C_i}}, \mathsf{P_{C_j}}) \tag{6.2}$$

where $\mathsf{P_{C_i}}$ and $\mathsf{P_{C_j}}$ represent the centroid (substring, cardinality) pairs for clusters $\mathsf{C_i}$ and $\mathsf{C_j}$ respectively. For the cluster $\mathsf{C_i}$, its centroid is denoted as $\mathsf{P_{C_i}} = (sub_{c_i} : card_{c_i})$. Here, $card_{c_i}$ is calculated as $\frac{1}{|C_i|}\sum_{j=1}^{|C_i|} card_{ij}$, where $card_{ij}$ refers to the actual cardinality of the $j$-th substring in $C_i$, $|C_i|$ is the size of the $i$-th cluster. And $sub_{c_i}$ is calculated as the median string of the $i$-th cluster. To calculate $sub_{c_i}$, we arrange all strings in the cluster $C_i$ in lexicographical order (alphabetical order for strings), and use $\lfloor \frac{|C_i|}{2} \rfloor$-th string in $C_i$ after the sorting to represent $sub_{c_i}$. The term $\mathsf{Dist}^2(\mathsf{P_{C_i}}, \mathsf{P_{C_j}})$ is the cardinality-distance between these centroids $\mathsf{P_{C_i}}$ and $\mathsf{P_{C_j}}$. Equation 6.2 calculates the increase in the sum of squared errors (SSE) upon merging clusters $\mathsf{C_i}$ and $\mathsf{C_j}$. In the hierarchical clustering process, clusters with the minimal $\Delta\mathsf{SSE}(\mathsf{C_i}, \mathsf{C_j})$ value are merged into a single cluster. **Step4:** Update the distances between the newly created cluster and the remaining clusters using Equation 6.2. Repeat steps 2, 3, and 4 until all (substring, cardinality) pairs in $Sub(DB)$ are unified into a single cluster. This methodology ensures minimal cardinality disparities within clusters, and significant variations between different clusters. The middle part of Figure 6.3 shows an example

of hierarchical clustering. In this example, three clusters are built, with each cluster represented by nodes of the same color.

### 6.3.3 Cluster-Centroid Embedding

ArbiLIKE introduces a multi-tiered contrastive embedding strategy to enable cardinality-aware embeddings for substrings in $Sub(DB)$. This strategy encompasses two embedding stages: inter-cluster contrastive embedding (Inter-CCE) and intra-cluster contrastive embedding (Intra-CCE). Inter-CCE aims to produce embeddings for each cluster's centroid, whereas Intra-CCE derives embeddings for substrings with each cluster, drawing upon the results from Inter-CCE. Next, we detail the cluster-centroid embedding by Inter-CCE.

ArbiLIKE includes a cluster-centroid embedding model, $\mathcal{M}_{\text{cent}}$, we borrow the idea proposed in contrastive learning method [100, 101] to build our cluster-centroid embedding model, $\mathcal{M}_{\text{cent}}$. The contrastive learning compares three data types: anchor, positive, and negative, within the embedding space. Specifically, $\mathcal{M}_{\text{cent}}$ is structured as a Triplet Network [102], which is designed for comparative analysis between these three data types. This network comprises three sub-networks, each dedicated to processing one data type. Within $\mathcal{M}_{\text{cent}}$, each sub-network is designed as a two-layer perceptron (See hyper-parameters tuning for details (§6.6.5)).

**Build the training dataset for centroid embedding.** To train the cluster-centroid embedding model $\mathcal{M}_{\text{cent}}$, the first step is building an appropriate training dataset. In ArbiLIKE, each (substring, cardinality) pair in $Sub(DB)$ serves as a data sample. The training dataset is built by choosing anchors, positive, and negative substrings from the previously constructed hierarchical clusters (§6.3.2).

Anchors selection. Anchors act as reference substrings against which positive and negative substrings are compared. From the hierarchical clusters established on $Sub(DB)$, centroids of these clusters are selected as anchors. The anchor for the $i$-th cluster corresponds to its centroid, denoted as $\mathbf{Anchor}^{\text{i}} = (sub_{c_i} : card_{c_i})$.

Positive substrings selection. Positive substrings are those closely resembling the anchor in terms of cardinality. The aim is to ensure that the embeddings of the anchor and these substrings are close to each other in the embedding space. For a given anchor $\mathbf{Anchor}^{i}$ of the $i$-th cluster, we utilize the $K$-nearest neighbour method

to identify the $K$ closest (substring, cardinality) pairs within the same cluster. Here, $K = N_c - 1$, with $N_c$ being the total number of cardinality-distance oriented clusters (§6.3.2). The $K$ nearest pairs of (substring, cardinality) are determined by the cardinality-distance and are chosen as positive substrings, represented as $\mathbf{Sub}^{\mathrm{i}}_{+}$.

Negative substrings selection. Negative substrings have notably different cardinalities from the anchor. The objective is to separate the embeddings of the anchor and these negative substrings in the embedding space. In ArbiLIKE, for a given anchor $\mathbf{Anchor}^{\mathrm{i}}$ of the $i$-th cluster, centroids from the other $N_c - 1$ clusters are selected as negative substrings. As $\mathbf{Anchor}^{\mathrm{i}}$ and these centroids come from different clusters, they inherently have different cardinalities. The set of negative substrings is denoted as $\mathbf{Sub}^{\mathrm{i}}_{-}$.

Training dataset. For the $i$-th cluster of (substring, cardinality) pairs, after determining the anchors, positive, and negative substrings, we denote the training dataset for embedding the cluster's centroid as $\mathbf{D}_{\mathrm{i}}$. This dataset comprises: $\mathbf{D}_{\mathrm{i}}$=($\mathbf{Anchor}^{\mathrm{i}}$, $\mathbf{Sub}^{\mathrm{i}}_{+}$, $\mathbf{Sub}^{\mathrm{i}}_{-}$), where $\mathbf{Anchor}^{\mathrm{i}}$ is the centroid, while $\mathbf{Sub}^{\mathrm{i}}_{+}$ and $\mathbf{Sub}^{\mathrm{i}}_{-}$ are the respective positive and negative substrings for the $i$-th cluster. Each training entry is a triplet ($\mathsf{s}^a$, $\mathsf{s}^p$, $\mathsf{s}^n$), derived from the $i$-th cluster's ($\mathbf{Anchor}^{\mathrm{i}}$, $\mathbf{Sub}^{\mathrm{i}}_{+}$, $\mathbf{Sub}^{\mathrm{i}}_{-}$).

**Cluster-centroid embedding.** In ArbiLIKE, we use the triplet ranking loss to learn embeddings for each cluster's centroid. For a triplet ($\mathsf{s}^a$, $\mathsf{s}^p$, $\mathsf{s}^n$) in $\mathbf{D}_{\mathrm{i}}$, the loss function is formulated as below.

$$\mathsf{Loss}(\mathsf{s}^{\mathsf{a}}, \mathsf{s}^{\mathsf{p}}, \mathsf{s}^{\mathsf{n}}) = \max(0, \mathrm{m}^{\mathrm{i}}_{\mathrm{inter}} + \mathrm{d}(\mathrm{E}^{\mathrm{a}}, \mathrm{E}^{\mathrm{p}}) - \mathrm{d}(\mathrm{E}^{\mathrm{a}}, \mathrm{E}^{\mathrm{n}})) \tag{6.3}$$

where $\mathrm{m}^{\mathrm{i}}_{\mathrm{inter}}$ is a margin chosen to distinguish the embedding distances of $(s^a, s^p)$ and $(s^a, s^n)$ for a given triplet $(s^a, s^p, s^n)$ in $\mathbf{D}_{\mathrm{i}}$. The embeddings $\mathrm{E}^{\mathrm{a}}$, $\mathrm{E}^{\mathrm{p}}$, and $\mathrm{E}^{\mathrm{n}}$ correspond to $\mathsf{s}^a$, $\mathsf{s}^p$, and $\mathsf{s}^n$ respectively and are updated during the training of the model $\mathcal{M}_{\mathrm{cent}}$. The terms $\mathrm{d}(\mathrm{E}^{\mathrm{a}}, \mathrm{E}^{\mathrm{p}})$, $\mathrm{d}(\mathrm{E}^{\mathrm{a}}, \mathrm{E}^{\mathrm{n}})$ represent the Euclidean distances between the embeddings $\mathrm{E}^{\mathrm{a}}$ and $\mathrm{E}^{\mathrm{p}}$, $\mathrm{E}^{\mathrm{a}}$ and $\mathrm{E}^{\mathrm{n}}$, respectively. The objective of Equation 6.3 is to guarantee that the distance between the embeddings of an anchor and a negative substring, $\mathrm{d}(\mathrm{E}^{\mathrm{a}}, \mathrm{E}^{\mathrm{n}})$, exceeds that between the anchor and a positive substring, $\mathrm{E}^{\mathrm{a}}$ and $\mathrm{E}^{\mathrm{p}}$. Let's analyze the three cases presented in Equation 6.3 to better understand how to achieve accurate embeddings for the centroid of the $i$-th cluster.

- **Case1:** $\mathrm{d}(\mathrm{E}^{\mathrm{a}}, \mathrm{E}^{\mathrm{n}}) > \mathrm{d}(\mathrm{E}^{\mathrm{a}}, \mathrm{E}^{\mathrm{p}}) + \mathrm{m}^{\mathrm{i}}_{\mathrm{inter}}$. Here, the negative substring's embedding

is adequately separated from the anchor compared to the positive substring's embedding. As a result, the loss is zero, leading to no updates in the embeddings for the anchor, positive, and negative substrings.

- **Case2:** $\mathrm{d}(\mathrm{E}^{\mathrm{a}}, \mathrm{E}^{\mathrm{n}}) < \mathrm{d}(\mathrm{E}^{\mathrm{a}}, \mathrm{E}^{\mathrm{p}})$. Here, the negative substring's embedding is closer to the anchor than the positive substring's embeddings. Given the loss exceeds $\mathrm{m}^{\mathrm{i}}_{\mathrm{inter}}$, leading to updates in the embeddings for both the positive and negative substrings.

- **Case3:** $\mathrm{d}(\mathrm{E}^{\mathrm{a}}, \mathrm{E}^{\mathrm{p}}) < \mathrm{d}(\mathrm{E}^{\mathrm{a}}, \mathrm{E}^{\mathrm{n}}) < \mathrm{d}(\mathrm{E}^{\mathrm{a}}, \mathrm{E}^{\mathrm{p}}) + \mathrm{m}^{\mathrm{i}}_{\mathrm{inter}}$ . Here, while the negative substring's embedding is more distant from the anchor than the positive substring's embedding, the distance doesn't exceed $\mathrm{m}^{\mathrm{i}}_{\mathrm{inter}}$. Thus, the loss remains positive but is less than $\mathrm{m}^{\mathrm{i}}_{\mathrm{inter}}$, resulting in updates only to the embeddings of the negative substrings.

**Selection of $\mathrm{m}^{\mathrm{i}}_{\mathrm{inter}}$.** In Equation 6.3, the margin $\mathrm{m}_{\mathrm{inter}}$ serves to balance the distance between positive pairs $\mathrm{d}(\mathrm{E}^{\mathrm{a}}, \mathrm{E}^{\mathrm{p}})$ and negative pairs $\mathrm{d}(\mathrm{E}^{\mathrm{a}}, \mathrm{E}^{\mathrm{n}})$. It ensures that the embeddings are well-separated while still being able to learn meaningful features. A small $\mathrm{m}^{\mathrm{i}}_{\mathrm{inter}}$ may inadequately differentiate positive and negative substring embeddings. Conversely, a large $\mathrm{m}^{\mathrm{i}}_{\mathrm{inter}}$ struggles to generate informative embeddings by compelling them to be overly distant.

In ArbiLIKE, we determine $\mathrm{m}^{\mathrm{i}}_{\mathrm{inter}}$ as follows. After initializing embedding for each triplet in $\mathbf{D}_{\mathrm{i}}$ prior to training model $\mathcal{M}_{\mathrm{cent}}$, we compute distances between embeddings of the anchor and all substrings in $\mathbf{Sub}^{\mathrm{i}}_{+}$ and $\mathbf{Sub}^{\mathrm{i}}_{-}$. Then, these distances are ranked in ascending order, and the distance at the $|\mathbf{Sub}^{\mathrm{i}}_{+}|$-th position is chosen as the value of $\mathrm{m}^{\mathrm{i}}_{\mathrm{inter}}$. This ensures the distance of every positive pair ($\mathsf{s}^a$, $\mathsf{s}^p$) remains less than all negative pairs ($\mathsf{s}^a$, $\mathsf{s}^n$). Note, $|\mathbf{Sub}^{\mathrm{i}}_{+}|$ represents the count of (substring, cardinality) pairs in $\mathbf{Sub}^{\mathrm{i}}_{+}$.

For each cardinality-distance oriented cluster (§6.3.2), we build a training dataset and learn the embedding of the cluster-centroid. The outcome of cluster-centroid embedding is a matrix, $\mathbf{E}_{\mathrm{cent}} \in \mathbf{R}^{N_c \times \lambda}$, where $N_c$ is the number of cluster centroids, and $\lambda$ is the embedding size. The left part of Figure 6.3 shows the three main steps to obtain the centroid embeddings. First, ArbiLIKE constructs the training dataset for each cluster, comprising numerous triples (Anchors, Positives, Negatives). In our

example, we consider three clusters. Second, we assign initial embeddings for training data samples, and identify the positive and negative pairs for each training data sample. Finally, ArbiLIKE employs the Inter-CCE approach to obtain embeddings for each centroid in the hierarchical clusters (§6.3.2). These results are used in the Intra-CCE process (§6.3.4) to acquire embeddings for the collected substrings (§6.3.1) in ArbiLIKE.

### 6.3.4 Cardinality-Aware Substrings Embedding

We illustrate how the Intra-CCE method converts substrings into feature vectors. ArbiLIKE incorporates a substrings embedding model $\mathcal{M}_{\text{sub}}$ which is adapted from the contrastive learning technique for the substrings embedding. $\mathcal{M}_{\text{sub}}$ is a Triplet Network, similar to the structure of cluster-centroid embedding model $\mathcal{M}_{\text{cent}}$ (§6.3.3). We use AutoML [103] to find the optimal structure of the $\mathcal{M}_{\text{sub}}$ (§6.6.5). Before training $\mathcal{M}_{\text{sub}}$, we need to build the training dataset.

**Build training dataset for substrings embedding.** For substring embedding within a specific cluster, we construct a training dataset comprising triples denoted as (**Anchor**, $\mathbf{Sub}_{+}^{\text{intra}}$, $\mathbf{Sub}_{-}^{\text{intra}}$), where the **Anchor** denotes the cluster's centroid. Within the cluster, we categorize all (substring, cardinality) pairs into two subsets: positive and negative pairs. ArbiLIKE utilizes a max-margin selection strategy [104] to optimally partition the pairs of (substring, cardinality) into two distinct groups. The loss function for $\mathcal{M}_{\text{sub}}$ aligns with Equation 6.3. Here, the margin for the Intra-CCE method is denoted as $\text{m}_{\text{intra}}$, its value is chosen based on the distance between the optimal boundary and the cluster centroid.

**Intra-cluster contrastive pairs.** In ArbiLIKE, using the margin $\text{m}_{\text{intra}}$, (substring, cardinality) pairs within a cluster are divided into two groups, $\mathbf{Sub}_{+}^{\text{intra}}$ and $\mathbf{Sub}_{-}^{\text{intra}}$. Substrings in $\mathbf{Sub}_{+}^{\text{intra}}$ exhibit smaller cardinality-distance to the centroid than those in $\mathbf{Sub}_{-}^{\text{intra}}$. The training dataset for substrings embeddings within a cluster, represented as $\mathbf{D}_{\text{train}}$, consists of triples ($\mathsf{s}^a$, $\mathsf{s}^p$, $\mathsf{s}^n$). Here, $\mathsf{s}^a$ is the cluster's centroid, while $\mathsf{s}^p$ and $\mathsf{s}^n$ are positive and negative substrings from $\mathbf{Sub}_{+}^{\text{intra}}$ and $\mathbf{Sub}_{-}^{\text{intra}}$, respectively.

**Substrings embedding.** Figure 6.4 depicts the substring embeddings generation process using Intra-CCE. Prior to the substring embeddings, each substring within the cluster is initialized with an embedding vector of length $\lambda$. As indicated in Figure 6.4,

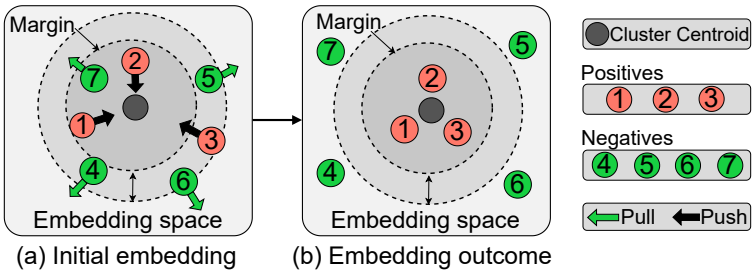


**Figure 6.4:** The intra-cluster contrastive embedding for the pairs of (substring, cardinality) within a cluster.

the positive substrings, $\mathbf{Sub}_{+}^{\text{intra}}$, consists of $\{sub_1, sub_2, sub_3\}$, while the negative substrings $\mathbf{Sub}_{-}^{\text{intra}}$, comprise $\{sub_4, sub_5, sub_6, sub_7\}$. Initially, in the embedding space, the positive substring $sub_3$ lies beyond the intra-cluster margin, whereas the negative substring $sub_7$ falls within this margin (Figure 6.4(a)). Upon applying Intra-CCE, the embeddings of positive substrings converge closer to the centroid, whereas those embeddings of the negative substrings disperse farther from the centroid within the embedding space, as illustrated in Figure 6.4(b). The goal of Intra-CCE is the training of $\mathcal{M}_{\text{sub}}$ and the generation of embeddings for substrings in $Sub(DB)$. The resulting substrings embedding is a matrix $\mathbf{E}_{\text{sub}} \in \mathbf{R}^{N_{sub} \times \lambda}$, where $N_{sub}$ denotes the count of substrings in $Sub(DB)$, and $\lambda$ is the embedding size.

## 6.4 Sequence model-based estimator

In this section, we introduce the proposed substring-based sequence model in ArbiLIKE. We introduce a novel sequence model-based estimator.

### 6.4.1 Cardinality Estimation via Sequence Model

This section shows how to leverage the character based language model for the cardinality estimation. The string in a LIKE predicate can be decomposed into a sequence of substrings. In ArbiLIKE, we use a substring-importance boosted sequence model to estimate the cardinalities of LIKE predicates.

**Substring-based sequence model.** Given a string $S$, by applying the $q$-grams technique, we obtain all the substrings of $S$ with a length of $q$. Consequently, $S$ can be represented as a sequence of (possibly) overlapped substrings: $S =< sub_1, ..., sub_t >$, where $t$ is the number of decomposed substrings. The substring-based sequence model

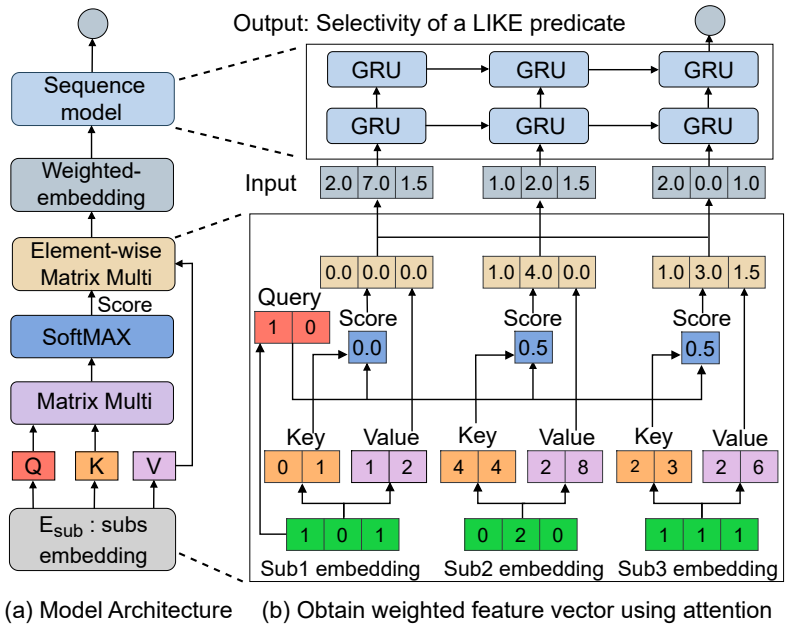


**Figure 6.5:** Cardinality estimator in ArbiLIKE.

is used to estimate the joint probability $P(S) = P(sub_1, sub_2, ..., sub_t)$ using the following formula.

$$P(S) = P(sub_1) \times \prod_{i=2}^{t} P(sub_t | sub_1, ..., sub_{i-1}) \tag{6.4}$$

where $t$ denotes the number of potentially overlapped substrings within $S$, and $P(S)$ represents the joint probability of the string $S$. Once this probability is computed, we use it to estimate the cardinalities of the LIKE predicates.

### 6.4.2 Substring-importance Boosted Model

As shown in left part of Figure 6.5, our substring-importance boosted sequence model $\mathcal{M}_{\text{est}}$ incorporates two parts: the first part is a self-attention mechanism [105] capturing the importance of each substring of string tuples in DB, and the second part is a Gated Recurrent Unit (GRU), a type of commonly used recurrent neural network (see details of the model structure in (§6.6.5)). We discuss $\mathcal{M}_{\text{est}}$ for cardinality estimation in this section.

**Training dataset generation.** To train the sequence model $\mathcal{M}_{\text{est}}$, we must build the training dataset first. Given a string database, $\mathsf{DB} = \{S_i\}_{i=1}^{N_S}$, the process of

generating training data has two steps: (1) We insert special characters '$\wedge$' and '\$' into each string tuple $S_i \in \mathsf{DB}$ to signify the start and end of $S_i$ respectively; (2) We then decompose $S_i$ into a sequence of substrings using the $q$-grams technique (§6.2). For instance, if $\mathsf{DB} = \{joe, jony\}$ and $q = 2$, the training dataset is represented as $\mathbf{D}_{\text{seq}} = \{\wedge j, jo, oe, e\$, \wedge j, jo, on, ny, y\$\}$. We use $\mathbf{\Sigma}_{\mathsf{sub}}$ to denote the set of unique substrings with a length of $q$ in $\mathsf{DB}$. In this example, $\mathbf{\Sigma}_{\mathsf{sub}} = \{\wedge j, jo, oe, e\$, on, ny, y\$\}$.

**Loss function for the estimator.** We describe the loss function $\mathsf{Loss}(\mathbf{D}_{\mathsf{seq}}, \mathbf{\Sigma}_{\mathsf{sub}})$ in Equation 6.5 for the substring-importance boosted sequence model $\mathcal{M}_{\text{est}}$. In $\mathcal{M}_{\text{est}}$, the loss function processes the substrings of each string tuple $S_i \in \mathsf{DB}$ ($i \in [1, N_S]$) sequentially one substring at a time, and seeks to accurately estimate the probability distribution for the next substring given the preceding substrings. For instance, in the running example above, the distinct substrings set $\mathbf{\Sigma}_{\mathsf{sub}} = \{\wedge j, jo, oe, e\$, on, ny, y\$\}$. When using "$jo, on$" as an input, $\mathcal{M}_{\text{est}}$ generates a probability distribution suggesting that the subsequent substring could be any in $\mathbf{\Sigma}_{\mathsf{sub}}$.

Based on the generated training dataset $\mathbf{D}_{\text{seq}}$ and the collected unique substrings $\mathbf{\Sigma}_{\mathsf{sub}}$ from the database $\mathsf{DB}$, we build the loss function $\mathsf{Loss}(\mathbf{D}_{\mathsf{seq}}, \mathbf{\Sigma}_{\mathsf{sub}})$. Let $\hat{y}^{(j)}$ and $y^{(j)}$ represent the probabilities of the predicated and actual next substring respectively. We use the cross-entropy (CE) [106] to quantify the loss between the probability distributions for $\hat{y}^{(j)}$ and $y^{(j)}$. The calculation of $\mathsf{Loss}(\mathbf{D}_{\mathsf{seq}}, \mathbf{\Sigma}_{\mathsf{sub}})$ is listed as below.

$$\mathsf{Loss}(\mathbf{D}_{\mathsf{seq}}, \mathbf{\Sigma}_{\mathsf{sub}}) = -\frac{1}{|\mathbf{D}_{\mathsf{seq}}|} \sum_{i=1}^{|\mathbf{D}_{\mathsf{seq}}|} \sum_{j=1}^{|\mathbf{\Sigma}_{\mathsf{sub}}|} y_i^{(j)} \log \hat{y}_i^{(j)} \tag{6.5}$$

where $\hat{y}_i^{(j)}$ and $y_i^{(j)}$ represent the predicted and actual probabilities, respectively, that the $i$-th substring in $\mathbf{D}_{\mathsf{seq}}$ occurs at the position $j$ in $\mathbf{\Sigma}_{\mathsf{sub}}$, $|\mathbf{D}_{\mathsf{seq}}|$ is the size of the training dataset, and $|\mathbf{\Sigma}_{\mathsf{sub}}|$ is the total number of unique substrings in $\mathbf{\Sigma}_{\mathsf{sub}}$. Equation 6.5 minimizes the cross entropy error of $\hat{y}_i^{(j)}$ and $y_i^{(j)}$ across the entire training datatset $\mathbf{D}_{\mathsf{seq}}$.

**Model training.** Algorithm 5 depicts the process of training the estimator $\mathcal{M}_{\text{est}}$. This algorithm produces the parameters required for the cardinality estimator, which subsequently can be used to estimate the cardinalities of LIKE predicates. The training dataset fed into Algorithm 5 is denoted as $\mathbf{D}_{\mathsf{seq}}$, comprising continuous substrings with a length of $q$. Additional inputs include a set of unique substrings, $\mathbf{\Sigma}_{\mathsf{sub}}$, the

**Algorithm 5** Substring-importance boosted estimator

**Input:** $\mathbf{D}_{\mathsf{seq}}$: Training dataset generated from $\mathsf{DB}$
$\mathbf{\Sigma}_{\mathsf{sub}}$: Set of unique substrings of $\mathsf{DB}$
$\mathbf{E}_{\mathsf{sub}}$: Substring embeddings (§6.3.4)
B: Batch size during the model training
$\alpha$: Learning rate when training model $\mathcal{M}_{\text{est}}$
$\lambda$: Substring embedding size (§6.3.4)
**Output:** Parameters of trained estimator $\Theta$

```
Initialize W_Q ∈ R^{λ×λ}, W_K ∈ R^{λ×λ}, W_V ∈ R^{λ×λ}, Θ
 for number of training iterations do
    for b < number of batches do
       D_b ← get_minibatch(D_seq, B, b)   E_b ← columns_mapping(E_sub, D_b)
       Q_b = E_b × W_Q, K_b = E_b × W_K, V_b = E_b × W_V
       W_imp ← soft_max(Q_b × K_b^T / √λ)
       W_emb = W_imp × V_b   Σ_b ← columns_mapping(Σ_sub, B, b)
       Loss(D_b, Σ_b, Θ) = −Log[Pr(Σ_b|W_emb)]
       Θ = Θ − α ∂Loss(D_b, Σ_b, Θ)/∂Θ
    end
end
return Θ;
```

substring embeddings $\mathbf{E}_{\mathsf{sub}}$ (§6.3.4), a predefined batch size $B$, a learning rate $\alpha$, and the size of substring embeddings $\lambda$. Before model training, we initialize three weight matrices, $\mathbf{W}_Q$, $\mathbf{W}_K$, and $\mathbf{W}_V$, each having dimensions $\mathbf{R}^{\lambda\times\lambda}$. These matrices are associated with the Query, Key, and Value matrices in the self-attention mechanism. During the model training, for each epoch (as described in Lines 2-11) and each batch (Lines 3-11), the algorithm first fetches a mini-batch $\mathbf{D}_{\mathsf{b}}$ of data instances, and each instance is a substring in $\mathbf{D}_{\mathsf{seq}}$ (Line 4). Then, the algorithm extracts the corresponding embeddings for the fetched substrings, denoted as $\mathbf{E}_{\mathsf{b}} \in \mathbf{R}^{B\times\lambda}$, with $B$ representing the batch size and $\lambda$ is the size of substring embedding (Line 5). Utilizing $\mathbf{E}_{\mathsf{b}}$, we calculate the Query ($\mathbf{Q}_{\mathsf{b}}$), Key ($\mathbf{K}_{\mathsf{b}}$), and Value ($\mathbf{V}_{\mathsf{b}}$) matrices (Line 6). Once we obtain the $\mathbf{Q}_{\mathsf{b}}$, $\mathbf{K}_{\mathsf{b}}$, and $\mathbf{V}_{\mathsf{b}}$ matrices, the algorithm computes the importance score for each substring in $\mathbf{D}_{\mathsf{b}}$, this yields the weighted-embedding matrix $\mathbf{W}_{\mathsf{emb}} \in \mathbf{R}^{B\times\lambda}$ (Lines 7-8). Finally, using $\mathbf{W}_{\mathsf{emb}}$ as the input for the loss function (Equation 6.5), the parameters $\Theta$ are learnt through the process outlined in Lines 10-11. Algorithm 5 thus yields the trained model parameters $\Theta$ for $\mathcal{M}_{\text{est}}$ (Line 12). Figure 6.5 provides an example of applying ArbiLIKE to a string tuple in $\mathsf{DB}$ with three substrings.

## 6.5 Extension to Generic LIKE Predicates

In this section, we illustrate how ArbiLIKE addresses the challenge of LIKE predicates with arbitrary number of wildcards("%","_").

### 6.5.1 Formulation as a Set Resemblance Problem

We formulate the cardinality estimation of arbitrary LIKE predicates as a set resemblance problem. A typical LIKE predicate may contain multiple wildcards ("%", "_") which partitions the string in the LIKE predicate into various segments.

For a generic LIKE predicate "LIKE $\%\text{seg}_1\%\text{seg}_2\%, ..., \%\text{seg}_m\%$", each $\text{seg}_i$ is a sequence of characters with a length of $\ell_{\text{seg}_i}$, where $i \in [1, m]$. The inverted list (§6.2) of $\text{seg}_i$ is defined as $\mathbf{I}_i$. In practice, collecting inverted lists for $\text{seg}_i$ of any length is unrealistic due to the considerable time and space overheads. In ArbiLIKE, if $\ell_{\text{seg}_i} > q$, we identify all $\text{seg}_i$'s substrings (max length of $q$), and select the substring with the smallest cardinality. We then utilize the inverted list associated with this selected substring to approximate the $\text{seg}_i$'s inverted list. We use $\mathbf{P}_{\text{seg}} = \{\text{seg}_i\}_{i=1}^{m}$ to denote a set of string segments which can be induced from "$\%\text{seg}_1\%\text{seg}_2\%, ..., \%\text{seg}_m\%$". If the resemblance value among the inverted lists for string segments in $\mathbf{P}_{\text{seg}}$ is known, the cardinality of a generic LIKE predicate "LIKE $\%\text{seg}_1\%\text{seg}_2\%, ..., \%\text{seg}_m\%$" is estimated using Equation 6.6.

$$\mathsf{Card}(\mathbf{P}_{\text{seg}}) = \{\max_{1 \le i \le m} \mathsf{Card}(\text{seg}_i)\} \times \mathbf{Res}(\mathbf{P}_{\text{seg}}) \tag{6.6}$$

where $\max \mathsf{Card}(\text{seg}_i), 1 \le i \le m$, denotes the maximum estimated cardinality among all the string segments within $\mathbf{P}_{\text{seg}}$. Each $\mathsf{Card}(\text{seg}_i)$, where $i \in [1, m]$, is estimated by the model $\mathcal{M}_{\text{est}}$ (§6.4.2). The term $\mathbf{Res}(\mathbf{P}_{\text{seg}})$ represents the resemblance value of the inverted lists for all string segments in $\mathbf{P}_{\text{seg}}$. The value of $\mathbf{Res}(\mathbf{P}_{\text{seg}})$ lies within $[0, 1]$, and $\mathbf{Res}(\mathbf{P}_{\text{seg}})$ quantifies the similarity of these inverted lists. In Equation 6.6, $\mathbf{Res}(\mathbf{P}_{\text{seg}})$ is computed using Equation 6.7.

$$\mathbf{Res}(\mathbf{P}_{\text{seg}}) = \frac{|\mathbf{I}_1 \cap \mathbf{I}_2, ..., \cap \mathbf{I}_m|}{|\mathbf{I}_1 \cup \mathbf{I}_2, ..., \cup \mathbf{I}_m|} \tag{6.7}$$

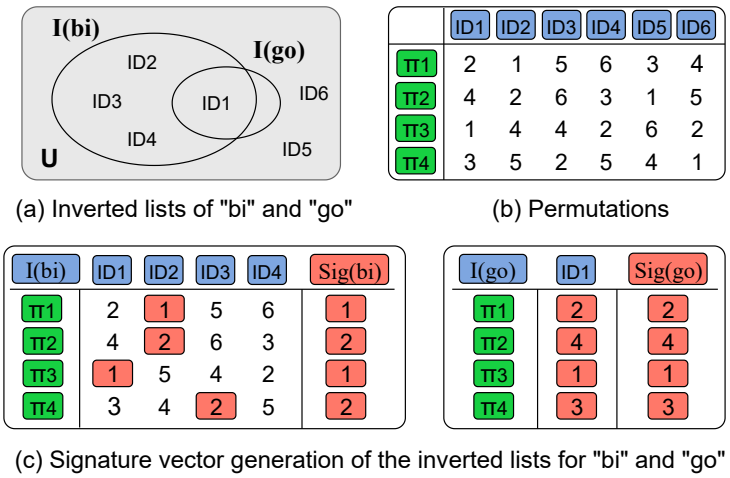


**Figure 6.6:** Illustration of generating the signature vectors of the inverted lists. (a) Inverted lists of **I**(bi) and **I**(go). (b) The permutations of the string IDs. The first row is the string IDs, and the first column represents four random permutations of the string IDs. (c) The signature vectors of **I**(bi) and **I**(go).

### 6.5.2 Signature Vector of Inverted List

Some string segments in $\mathbf{P}_{\text{seg}}$ may appear in a large portion of string tuples in DB, which results in long inverted lists and an unacceptable time complexity to perform intersection and union operations in Equation 6.6. To address this problem, ArbiLIKE uses an efficient approach to calculate the set resemblance which is adapted from the existing methods [107, 108]. In ArbiLIKE, we must calculate the set resemblance in an efficient way.

ArbiLIKE uses a *signature vector* to approximately represent an inverted list to calculate the set resemblance for efficiency. The signature vector is a real-valued vector with a fixed length and keeps the characteristics of a large set. ArbiLIKE uses a Monte Carlo approximation technique to build a signature vector of an inverted list. The signature vector of $\mathbf{I}_i$ is denoted as $\mathbf{Sig}_i = (X_i^1, X_i^2, ..., X_i^L)$, where each element $X_i^j$ is a real valued number, $i \in [1, m]$, $j \in [1, L]$, $m$ is the number of different segments in a generic LIKE predicate, and $L$ is the length of the signature vector.

We use $\mathsf{U} = \{1, ..., N_S\}$ to denote the universe of string tuple IDs in DB where $N_S$ is the number of string tuples in DB. Given a generic LIKE predicate in the form of "%$\text{seg}_1$%$\text{seg}_2$%, ..., %$\text{seg}_m$%", and the inverted list of $i$-th segment is $\mathbf{I}_i$. Based on the definition of the inverted list (§6.2), we have $\mathbf{I}_i \subseteq \mathsf{U}$. Assume that $\pi$ is a permutation of U. We define $\min\{\pi(\mathbf{I}_i)\} = \min\{\pi(x) | x \in \mathbf{I}_i\}$. We select $\pi_1, ..., \pi_L$, which represent

$L$ uniform random permutations derived from $\mathsf{U}$. For an inverted list $\mathbf{I}_i$ of $i$-th string segment in a generic LIKE predicate, we define its signature vector as follows.

$$\mathbf{Sig}_i = (min\{\pi_1(\mathbf{I}_i)\}, min\{\pi_2(\mathbf{I}_i)\}, ..., \min\{\pi_L(\mathbf{I}_i)\}) \tag{6.8}$$

where term $min\{\pi_j(\mathbf{I}_i)\}$ is the minimum value of the $j$-th permutation ($j \in [1, L]$) and $L$ is the total count of permutations applicable to string tuple IDs in $\mathsf{DB}$. Hence, for the $j$-th element $X_i^j$ in $\mathbf{Sig}_i$, we have $X_i^j = min\{\pi_j(\mathbf{I}_i)\}$. In practice, to get the proper value of each element in a signature vector, we independently seed a hash function and generate a hash value $h(x)$ for each element $x \in \mathbf{I}_i$; the minimum $h(x)$ is recorded in the signature vector. In ArbiLIKE, we employ linear hash functions [109], because of their efficacy and ease of generating numerous independent hash functions. Each hash function is selected to ensure a low probability of collision.

We use $\widehat{\mathbf{Res}}(\mathbf{P}_{\text{seg}})$ to represent an approximated value of the $\mathbf{Res}(\mathbf{P}_{\text{seg}})$ (§6.5.1), where $\mathbf{Res}(\mathbf{P}_{\text{seg}})$ represents the resemblance value of the inverted lists for all string segments in a generic LIKE predicate, "LIKE $\%\text{seg}_1\%\text{seg}_2\%, ..., \%\text{seg}_m\%$". we calculate $\widehat{\mathbf{Res}}(\mathbf{P}_{\text{seg}})$ using the following formula.

$$\widehat{\mathbf{Res}}(\mathbf{P}_{\text{seg}}) = \frac{|\{i|\, \min\{\pi_i(\mathbf{I}_1)\} = \cdots = \min\{\pi_i(\mathbf{I}_m)\}\}|}{L} \tag{6.9}$$

where $L$ is the number of hash functions and $L$ is equal to the length of a signature vector. With the appropriate selection of $L$, $\widehat{\mathbf{Res}}(\mathbf{P}_{\text{seg}})$ is guaranteed to provide a reliable approximation of $\mathbf{Res}(\mathbf{P}_{\text{seg}})$ [110]. Figure 6.6 gives an example of calculating the signature vectors of two inverted lists: $\mathbf{I}$(bi) and $\mathbf{I}$(go). These two lists are derived from the Customer table in Figure 6.2(a). We use the following four steps to generate signature vectors for $\mathbf{I}$(bi) and $\mathbf{I}$(go).

**Step1**: get the entire string IDs and inverted lists of $\mathbf{I}$(bi) and $\mathbf{I}$(go). In this example, $\mathbf{I}$(bi) = $\{1, 2, 3, 4\}$ and $\mathbf{I}$(go) = $\{1\}$. The entire string tuple IDs are represented as IDs = $\{1, 2, 3, 4, 5, 6\}$. To make the example clearer, we use ID1, ..., ID6 to denote string tuple ID from 1 to 6 in the Customer table (Figure 6.6(a)).

**Step2**: get $L$ random permutations for the entire string tuple IDs. The signature vector length is equal to the number of random permutations. In this example, we set $L = 4$, and the four random permutations are $\pi_1$, $\pi_2$, $\pi_3$, and $\pi_4$ (Figure 6.6(b)).

**Step3**: determine the signature vectors for the inverted lists $\mathbf{I}$(bi) and $\mathbf{I}$(go).

For each random permutation $\pi_i$, where $i \in [1, 4]$, we extract the minimum value in $\pi_i$, and this value becomes the $i$-th element of the signature vector. After iterating through all the random permutations, we obtain the signature vector corresponding to an inverted list. As Figure 6.6(c) shows, for $\mathbf{I}$(bi), $\pi_1$ maps ID1, ID2, ID3, and ID4 to 2, 1, 5, 6 respectively. Thus, the minimum value of $\pi_1$ is 1. Similarly, we find the minimum mapped values for the remaining random permutations. The resulting signature vectors for $\mathbf{I}$(bi) and $\mathbf{I}$(go) are denoted as $\mathbf{Sig}$(bi) and $\mathbf{Sig}$(go) respectively. In this instance, $\mathbf{Sig}$(bi) =is {1,2,1,2} while $\mathbf{Sig}$(go) is {2,4,1,3}.

**Step4**: get the approximation of a set resemblance value of $\mathbf{Sig}$(bi) and $\mathbf{Sig}$(go) based on Equation 6.9, which is computed as 0.25. We utilize this value to estimate the cardinality of "LIKE %bi%go%" based on Equation 6.6. And the estimated cardinality is 1, which is the same as the actual cardinality of "LIKE %bi%go%".

### 6.5.3 Extend to Multiple Columns

As a table may contain multiple string columns, it is common that a LIKE predicate is constructed based on more than one string columns. In this section, we introduce how to make ArbiLIKE handle LIKE predicates involves different columns by leveraging the set join and union operations.

**Different columns.** Assuming a table with at least two string columns, Col1 and Col2, we examine two basic types of LIKE predicates based on "AND" and "OR", which can be extended to more complex predicates. (1) Different Columns with "AND": Predicates can be in the form of "Col1 LIKE %seg_1% AND Col2 LIKE %seg_2%", where "%seg_1%" and "%seg_2%" are sequences of characters. **Step 1**: Obtain signature vectors for %seg_1% and %seg_2% (denoted as $\mathbf{Sig_1}$ and $\mathbf{Sig_2}$, representing the rows in the table containing these character sequences. The signature vectors convey the information of which rows in the table contain the corresponding character sequence. **Step 2**: Calculate the similarity of the signature vectors, $Sim(\mathbf{Sig_1}, \mathbf{Sig_2}) \in [0, 1]$, using cosine similarity. This value approximates the ratio of rows containing both "%seg_1%" and "%seg_2%". **Step 3**: Approximate the results using $Sim(\mathbf{Sig_1}, \mathbf{Sig_2})$. Once the cardinalities for "%seg_1%" and "%seg_2%" (represented as $\mathbf{Card_1}$ and $\mathbf{Card_2}$) are obtained, calculate the results as: $Min(\mathbf{Card_1}, \mathbf{Card_2}) \times Sim(\mathbf{Sig_1}, \mathbf{Sig_2})$ (similar to set join operations). (2) Different

Columns with "OR": The process is similar to the "AND" case, with a variation in Step 3 for calculating results. **Step 3** (OR Variant): Calculate the results as: $Max(\mathbf{Card_1}, \mathbf{Card_2}) + Min(\mathbf{Card_1}, \mathbf{Card_2}) \times Sim(\mathbf{Sig_1}, \mathbf{Sig_2})$. This formula resembles the calculation used in set union operations.

**Predicates with numeric columns.** In handling predicates with numeric and string columns, ArbiLIKE employs a structured approach. **Step 1.** Selectivity estimation for numeric columns: Using an autoregressive model, ArbiLIKE estimates the selectivity ($Sel_1$) of numeric conditions (like Col1 $\leq v_1$), representing the proportion of rows that meet this criterion. **Step 2.** Cardinality estimation for LIKE operator: For string column predicates, ArbiLIKE calculates the cardinality ($Card_2$), indicating the number of rows that match the LIKE condition. **Step 3.** Combining for "AND" operation: For combined "AND" predicates (e.g., Col1 $\leq v_1$ AND Col2 LIKE %seg%), ArbiLIKE approximates the final cardinality by multiplying $Sel_1$ and $Card_2$. **Step 4.** Combining for "OR" operation: For "OR" predicates (e.g., Col1 $\leq v_1$ OR Col2 LIKE %seg%), it calculates the result using: $Max(Card_1, Card_2) + (1 - Sel_1) \times Min(Card_1, Card_2)$, where $Card_1$ is the cardinality of the numeric condition. This method allows ArbiLIKE to effectively handle queries with both numeric and string columns, enhancing its capability in database query optimization.

**Predicates containing the "_" wildcard.** Given that the wildcard "_" represents a single character in a string, our first step is using $\mathcal{M}_{est}$ to predict potential characters that can substitute the "_". Each "_" is then replaced with the predicted character. This transforms a predicate containing the "_" wildcard into a form resembling LIKE "%$\text{seg}_1$%$\text{seg}_2$%, ..., %$\text{seg}_m$%". Subsequently, we adopt the method presented in Section 6.5.1 to estimate the cardinalities for predicates with the "_" wildcard.

## 6.6 Evaluation

We compare ArbiLIKE with state-of-the-art cardinality estimators for LIKE predicates. We aim to answer the following questions:

- How does ArbiLIKE perform in terms of accuracy when dealing with arbitrary LIKE predicates? (§6.6.2 and §6.6.3)

**Table 6.2:** Statistics for the datasets used in ArbiLIKE.

| Source | Column | Abbrv | #Entries | Min Len | Max Len |
|---|---|---|---|---|---|
| DBLP | Article Titles | DBLP_AT | 50K | 3 | 127 |
| DBLP | Author Names | DBLP_AN | 82K | 2 | 64 |
| IMDB | Movie Notes | IMDB_MN | 3.66M | 2 | 147 |
| IMDB | Movie Titles | IMDB_MT | 4.56M | 4 | 85 |
| IMDB | Actor Names | IMDB_AN | 4.26M | 2 | 135 |

**Table 6.3:** Workloads used for evaluation. #Subs: Number of substrings. Wilds: Wildcards type contained in LIKE predicates. #Wilds: Number of wildcards in the workloads. Feature: Characteristic of LIKE predicates for each workload.

| Workloads | #Subs | Wilds | #Wilds | Feature |
|---|---|---|---|---|
| QS-base | 1 | "%" | 1 or 2 | Single-subs |
| QS-multi-subs | 2-5 | "%" | 3-6 | Multi-subs |
| QS-diff-wilds | 2-5 | "%", "_" | 3-6 | Diff-wildcards |

- How does the improvement on the cardinality estimation impact the performance of the query optimizer? (§6.6.4)
- How do different hyper-parameters impact the estimation accuracy of ArbiLIKE? (§6.6.5)
- How does each component of ArbiLIKE boost the overall cardinality estimation accuracy? (§6.6.10)

### 6.6.1 Experimental Setup

**Platform.** We use a machine with an NVIDIA A100 GPU and an Intel i9 CPU with 96GB RAM, and Pytorch 2.0.

**Workloads.** We use real-world datasets (Table 6.2): IMDB [23] and DBLP [24]. Both have a number of string attributes. We use three workloads (Table 6.3). Each contains 2,000 testing queries. We discuss the workloads as follows.

- QS-base: This workload is constructed by following the existing work [16, 111, 18, 19, 31, 20, 112], it comprises 2,000 LIKE predicates in the form of %$S$%, $S$%, and

**Table 6.4:** Estimation errors on the three group of LIKE predicates workloads over five different datasets.

| Type | Estimator | DBLP_AT | | | DBLP_AN | | | IMDB_MN | | | IMDB_MT | | | IMDB_AN | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 50th | 90th | 99th | 50th | 90th | 99th | 50th | 90th | 99th | 50th | 90th | 99th | 50th | 90th | 99th |
| QS-Base | Postgres | 6.5 | 63.3 | 417.0 | 6.7 | 55.4 | 417.0 | 12.6 | 159.2 | 417.0 | 10.8 | 125.6 | 417.0 | 10.5 | 139.0 | 417.0 |
| | MO | 12.4 | 88.5 | 1342.3 | 14.3 | 97.5 | 1225.3 | 18.8 | 192.4 | 3680.5 | 14.6 | 130.0 | 1844.6 | 16.3 | 142.8 | 2563.3 |
| | BayesNet | 14.2 | 165.9 | 3945.6 | 16.1 | 158.2 | 3474.2 | 23.4 | 428.6 | 4428.7 | 15.2 | 135.8 | 3222.1 | 20.8 | 361.5 | 3734.5 |
| | CRT | 11.5 | 75.8 | 788.7 | 14.2 | 96.4 | 655.9 | 17.6 | 179.3 | 1135.5 | 13.7 | 95.3 | 961.5 | 14.4 | 146.0 | 846.8 |
| | LBS | 8.2 | 49.3 | 384.1 | 7.3 | 78.6 | 262.3 | 11.7 | 223.5 | 768.3 | 7.2 | 108.5 | 744.4 | 15.3 | 193.7 | 734.6 |
| | DREAM | 5.4 | 44.8 | 466.5 | 6.4 | 61.8 | 278.6 | 7.1 | 176.5 | 683.3 | 6.4 | 86.6 | 1042.7 | 10.5 | 153.9 | 779.0 |
| | P-SPH | 7.3 | 39.5 | 544.6 | 7.6 | 68.4 | 382.7 | 10.6 | 185.4 | 1034.6 | 8.0 | 97.3 | 846.5 | 13.8 | 164.5 | 1088.6 |
| | Astrid | 2.4 | 15.7 | 106.4 | 2.7 | 33.5 | 94.8 | 6.8 | 89.4 | 182.1 | 4.6 | 58.6 | 154.3 | 5.3 | 76.5 | 176.9 |
| | **ArbiLIKE** | **1.6** | **9.2** | **23.9** | **1.7** | **9.8** | **21.4** | **2.1** | **14.2** | **45.6** | **1.7** | **10.8** | **27.5** | **1.8** | **12.7** | **38.1** |
| QS-multi-subs | Postgres | 10.3 | 96.2 | 417.0 | 9.4 | 78.5 | 417.0 | 13.3 | 222.9 | 417.0 | 11.6 | 184.2 | 417.0 | 12.9 | 208.5 | 417.0 |
| | MO | 19.5 | 304.9 | 1880.1 | 17.8 | 234.8 | 1688.2 | 37.0 | 1021.3 | 4280.6 | 24.4 | 756.4 | 2256.2 | 32.4 | 1087.6 | 3336.7 |
| | BayesNet | 26.6 | 517.3 | 4045.6 | 24.3 | 425.1 | 2774.3 | 34.4 | 416.5 | 5282.9 | 25.6 | 226.2 | 3536.3 | 26.6 | 331.5 | 4322.3 |
| | CRT | 30.2 | 190.2 | 1277.7 | 26.6 | 155.0 | 905.3 | 39.4 | 220.2 | 1555.7 | 22.4 | 143.5 | 1136.4 | 32.7 | 169.0 | 1283.8 |
| | LBS | 14.8 | 163.6 | 933.5 | 13.2 | 151.6 | 784.9 | 16.3 | 356.8 | 1450.2 | 18.0 | 186.7 | 1327.5 | 17.7 | 164.6 | 1348.4 |
| | DREAM | 6.8 | 123.1 | 634.8 | 5.3 | 90.4 | 573.2 | 10.5 | 146.9 | 893.1 | 9.1 | 159.8 | 873.1 | 12.1 | 118.9 | 915.2 |
| | P-SPH | 9.7 | 145.7 | 783.4 | 12.6 | 128.4 | 658.5 | 11.8 | 185.2 | 1208.4 | 14.2 | 173.4 | 1033.8 | 13.7 | 150.4 | 1353.6 |
| | Astrid | 2.9 | 70.2 | 266.0 | 3.1 | 58.5 | 202.8 | 9.3 | 129.9 | 838.0 | 6.5 | 64.2 | 534.8 | 9.4 | 85.7 | 646.4 |
| | **ArbiLIKE** | **2.1** | **17.1** | **85.2** | **2.2** | **15.3** | **66.7** | **2.5** | **24.7** | **120.6** | **2.4** | **20.6** | **106.3** | **2.3** | **21.4** | **117.9** |
| QS-multi-wilds | Postgres | 11.9 | 156.2 | 417.0 | 11.5 | 117.6 | 417.0 | 12.4 | 269.4 | 417.0 | 11.9 | 227.3 | 417.0 | 13.3 | 234.3 | 417.0 |
| | MO | 36.3 | 514.5 | 3242.5 | 35.7 | 315.5 | 2484.8 | 46.3 | 1721.2 | 3800.8 | 26.3 | 1188.5 | 2456.4 | 38.6 | 1420.5 | 2678.1 |
| | BayesNet | 34.1 | 981.3 | 4004.8 | 29.5 | 477.3 | 4550.2 | 48.2 | 686.6 | 5182.0 | 37.8 | 265.2 | 4225.8 | 44.4 | 470.8 | 4583.4 |
| | CRT | 34.0 | 297.2 | 390.4 | 31.3 | 200.5 | 278.9 | 45.3 | 459.8 | 2440.2 | 24.2 | 237.4 | 1683.8 | 41.7 | 308.4 | 1797.6 |
| | LBS | 18.6 | 314.6 | 428.9 | 16.0 | 178.8 | 436.5 | 37.6 | 540.3 | 1855.4 | 23.6 | 309.4 | 1578.5 | 25.3 | 277.3 | 2067.1 |
| | DREAM | 11.2 | 186.4 | 477.5 | 10.7 | 155.8 | 466.0 | 13.8 | 510.5 | 1753.4 | 16.0 | 278.7 | 1336.9 | 14.6 | 402.6 | 1448.8 |
| | P-SPH | 14.5 | 263.5 | 462.1 | 13.5 | 195.4 | 394.0 | 28.9 | 416.6 | 2190.0 | 18.3 | 412.7 | 1970.4 | 17.4 | 338.5 | 1984.2 |
| | Astrid | 4.3 | 92.6 | 379.8 | 3.9 | 74.5 | 343.8 | 14.7 | 148.1 | 1414.2 | 7.7 | 97.8 | 904.4 | 10.5 | 132.5 | 1107.5 |
| | **ArbiLIKE** | **2.2** | **19.3** | **104.3** | **1.9** | **18.5** | **82.9** | **2.8** | **29.7** | **221.6** | **2.5** | **24.6** | **198.3** | **2.6** | **25.4** | **217.3** |

$\%S$, where $S$ represents a character string with a length varying between 2 to 12 characters.

- QS-multi-subs: This workload contains general LIKE predicates which involve multiple substring. The format of predicates likes this: $\%\text{seg}_1\%\text{seg}_2\%, ..., \%\text{seg}_m\%$, with each "seg" representing one or more characters. We create this workload by first choosing a random string tuple from a DB and selecting a random number $\tau$ ($3 \leq \tau \leq$ tuple_length). We then remove $\tau$ characters randomly from the string tuple and insert 3 to 6 wildcards "%" among the remaining characters.

- QS-diff-wilds: This workload contains LIKE predicates with "%" and "_" wildcards, we randomly select a string tuple from the database and a number $\tau$ ($3 \leq \tau \leq$ tuple_length). From this tuple, we remove $\tau$ characters, and insert "%" and "_" wildcards in the remaining characters. We ensure that each LIKE predicate contains at least one "%" wildcard and one "_" wildcard.

**Evaluation metric.** To evaluate the accuracy of ArbiLIKE, we use a commonly used metric Q-error [16, 111, 18, 19]. The Q-error of a query is the multiplicative factor

an estimated cardinality deviates from a query's true cardinality: $max(\frac{estim}{actual}, \frac{actual}{estim})$. In ArbiLIKE, we report the median, 90th, and 99th percentile errors.

**Baselines.** We compare ArbiLIKE against a variety of representative cardinality estimators, including: (1) **Postgres** [25]: A genuine DBMS using 1D histograms and heuristics for cardinality estimation. (2) **Maximal Overlap** (MO [27]): Utilizes pruned count-suffix trees and implements a maximal overlap strategy among substrings to enhance estimation accuracy. (3) **BayesNet** [28]: A leaned method that leverages a Bayesian network for cardinality estimation, it depends on the conditional independent assumption across substrings extracted from DB string tuples. (4) **Cardinality Resemblance Technique** (CRT [29]): Aims to overcome underestimation in cardinality estimation for LIKE predicates by identifying critical substrings. CRT is capable of addressing LIKE predicates comparising multiple substrings, e.g., "LIKE $\%\text{seg}_1\%\text{seg}_2\%, ..., \%\text{seg}_m\%$". (5) **Lower Bound Estimation** (LBS [107]): This technique relies on the information stored in an extended $q$-grams, LBS utilizes signatures generated by set hashing techniques to estimate the overlaps among groups of substrings of the string in the LIKE predicates. (6) **Deep Cardinality Estimation of Approximate substring queries** (DREAM [113]): Takes a query string and a threshold as input, and approximately estimates the cardinality of the query within an edit distance using the long short-term memory (LSTM [114]). (7) **Positional Sequence Patterns-based Histogram** (P-SPH [115]): Aims to employ a positional sequence pattern-based histogram structure to estimate the selectivity of LIKE queries. (8) **Astrid** [30]: A deep learning-based approach for cardinality estimation in LIKE predicates. It integrates an embedding learner with a sequence model for cardinality estimation.

**Adapt baselines for evaluation.** To evaluate the eight baseline methods, adaptations were necessary since not all were originally designed for LIKE predicates involving multiple substrings or varied wildcards. Postgres, CRT, and LBS are equipped to handle LIKE predicates across all three workload types in Table 6.2. For MO, BayesNet, and P-SPH, they can handle queries that have same format as queries in QS-base and QS-multi-subs workloads (Table 6.2). To enable these three methods to handle LIKE predicate with wildcard "_", we directly replace the wildcard "_" with "%", this is the same as the method used in LBS to handle LIKE predicates with wildcard "_". In ArbiLIKE, only the approach introduced in section 6.5 is related

to handle generic LIKE predicates. To enable the two deep learning based methods (DREAM and Astrid) handle LIKE predicates with multiple substrings or different wildcards ("%" and "_"), we replace our proposed LIKE predicates encoding method (§6.3) and the sequence model (§6.4) with each of these two methods, while retaining the method introduced in section 6.5 as it is.

### 6.6.2 Estimation Quality

Table 6.4 shows that ArbiLIKE exceeds the baseline estimators. Postgres utilizes heuristic methods for estimating cardinalities with LIKE predicates. While it caps errors with heuristics, it struggles to accurately represent inter-relationships of substrings, resulting in significant errors, particularly for predicates with multiple wildcards. MO (Maximal Overlap) employs $q$-gram based estimations, assuming independence among substrings, which often leads to underestimation of cardinalities for LIKE predicates. In comparison, ArbiLIKE achieves an accuracy boost ranging from 7.8× to 80.7× over MO. BayesNet assumes conditional independence among substrings, a premise that frequently fails in real-world databases, leading to large errors, especially at higher quantiles. Compared to BayesNet, ArbiLIKE improves accuracy by 8.9× to 165.1×. CRT (Cardinality Ratio Technique): Estimates cardinalities by identifying shorter strings similar in frequency to a given string in LIKE predicates. Reliance on this method leads to considerable errors, especially for predicates involving long strings. ArbiLIKE outperforms CRT with an accuracy improvement of 3.7×to 35.0×.

ArbiLIKE also have higher estimation accuracy compared to some recent published methods. LBS (Label-Based Sampling) uses an extended q-gram table for cardinality estimation but faces challenges with LIKE predicates that include strings longer than the $q$-gram length, leading to low accuracy. LBS estimates the cardinality of an input LIKE predicate based on the cardinalities of the minimal base string of the string in a LIKE predicate. Since the base string with lengths larger than $q$ do not exist in the extended $q$-gram table, the LBS approximates their cardinalities by using MO [27]. Therefore, LBS suffers from low estimation accuracy for a LIKE predicate with long strings. Compared to LBS, ArbiLIKE achieves an accuracy improvement ranging from 4.2× to 18.2×. DREAM applies a trie-based dynamic programming

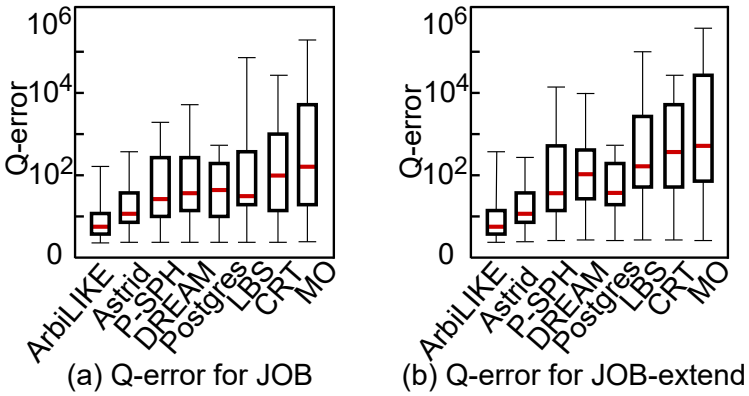


**Figure 6.7:** Cardinality estimation errors for the LIKE predicates within JOB and JOB-extend. The scale of the y-axis is logarithmic with base 10.

approach for fast query estimation. However, it heavily relies on a LSTM model for unrecorded strings, incurring high errors at the 99th percentile. P-SPH (Positional Sequence Pattern Histograms) extracts positional sequence patterns to build histograms for estimation. Its partial sequence matching approach, which uses short string cardinalities to represent longer ones, results in overestimation. Consequently, ArbiLIKE surpasses P-SPH in terms of accuracy ranging from 4.5× to 30.8×.

Astrid utilizes a deep sequence model for cardinality estimation but doesn't effectively recognize cardinality variations among substrings. Its character-driven sequence model fails to capture complex patterns within LIKE predicates, leading to errors beyond the 75th percentile, ArbiLIKE achieves up to 6.4× better accuracy than Astrid. Our method, ArbiLIKE, is presented as a new cardinality estimation for LIKE predicates. It achieves enhanced accuracy in cardinality estimation for LIKE predicates across all scenarios, particularly excelling where others fall short, such as with long strings and complex inter-string relationships.

### 6.6.3 Estimations on Standard Benchmarks

We evaluate the cardinality estimation accuracy ofor ArbiLIKE and baselines using two standard benchmarks: (1) the **Join Order Benchmark** (JOB) and (2) **JOB-extend.** JOB is based on IMDB dataset [23], it comprises 113 queries, with 80 containing LIKE predicates. Of these, 44 queries feature LIKE predicates tied to a single table and a single string column, while 4 queries pertain to a single table but two or more string columns. The remaining 32 queries incorporate LIKE predicates

associated with at least two tables and two string columns. JOB-extend is also a widely used benchmark for evaluation [116, 117, 118]. The queries in JOB-extend are derived from the same query templates as JOB. JOB-extend includes 2,240 queries in total, and 1,644 of them with LIKE predicates. Note that BayesNet was not evaluated on either JOB or JOB-extend due to its incompatibility with queries involving cyclic joins. This benchmark is used to test the scalability of ArbiLIKE on the query performance.

Figure 6.7 shows the cardinality estimation results of ArbiLIKE and baselines on the LIKE predicates within the queries of JOB and JOB-extend. Generally, the baselines tend to yield larger errors due to the broader cardinality space to be estimated for queries in JOB and JOB-extend compared to our workloads (Table 6.3). Compared to ArbiLIKE, Astrid, P-SPH, DREAM, Postgres, LBS, CRT, and MO exhibit up to 1.7× 2.1×, 7.4×, 3.5×, 9.8×, 11.2× and 19.5× greater median errors, respectively. The smallest estimation errors for all methods are observed in JOB's queries Q5a, Q5b, and Q5c, where the LIKE predicates involve only a single character, allowing for precise estimations by all methods. ArbiLIKE also exhibits the lowest variance in estimation errors, attributable to its effective LIKE predicate encoding (§6.3) and sequence model (§6.4), which capably learn underlying string patterns from the database columns, given sufficient training.

### 6.6.4 Impacts on Query Performance

We evaluate whether the improvement of cardinality estimation in ArbiLIKE leads to better query performance compared to baselines.

**Experimental setting.** To fairly evaluate the cardinality estimation results on execution plan, we use Postgres as common ground. We run all the methods (ArbiLIKE and the rest seven baselines) independently and collect their cardinality estimation results on LIKE predicates. Then, we inject these estimated cardinalities in a modified version of Postgres [77] and measure the runtime of each query.

**Workloads.** We use three workloads for the performance evaluation. (1) Ours. We use the workloads listed in Table 6.3 as the first benchmark for performance evaluation. (2) JOB (§6.6.3). We use the queries with LIKE predicates in JOB for another performance evaluation. (3) JOB-extend. We use this benchmark to

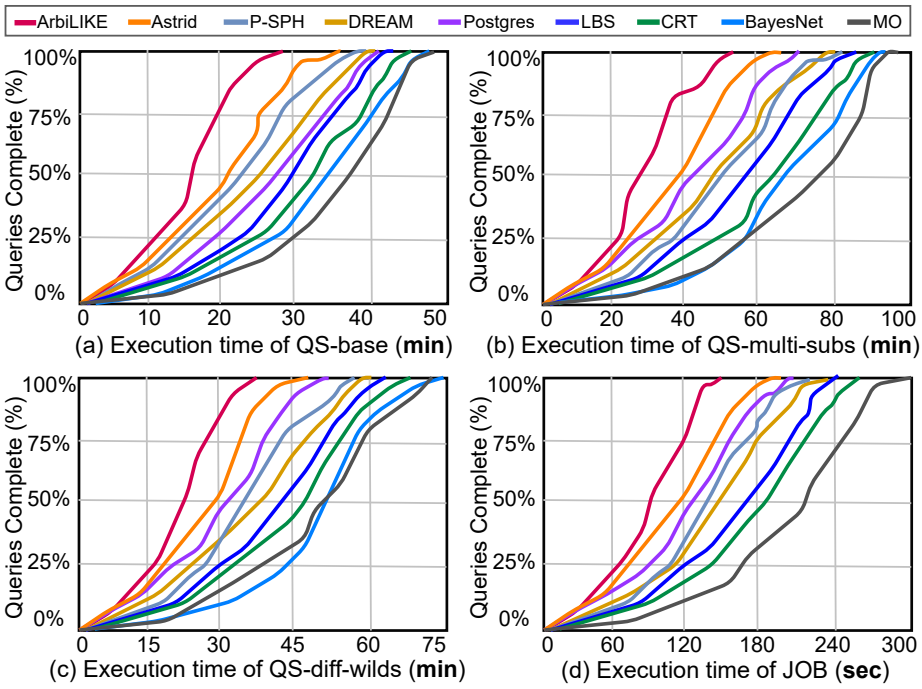


**Figure 6.8:** The percentage of queries completed throughout the runtime for both our workloads (Table 6.3) and the Join Order Benchmark (JOB).

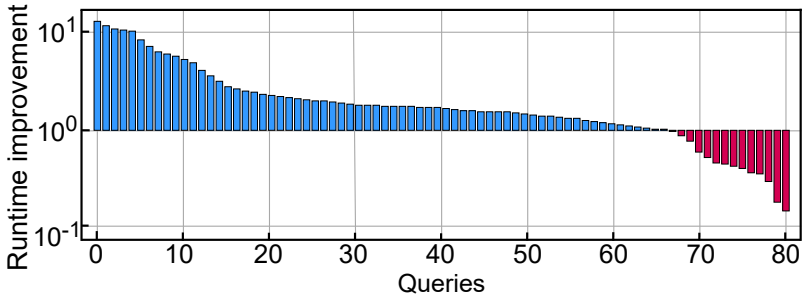


**Figure 6.9:** Relative runtime improvement for the JOB, where each bar represents a single query. These queries have been sorted from best to worst improvement. The scale of the y-axis is logarithmic with base 10.

test the scalability of different methods. and (3)JOB-extend (§6.6.3). We use a standard benchmark queries, JOB, for LIKE queries evaluation. JOB is based on IMDB dataset [23] with 113 queries, and 80 queries with LIKE predicates. Among these 80 queries, the LIKE predicates in 44 of them are based on one table one string column, 4 of them are based on one table two string columns. For the the rest of 32 queries, their LIKE predicates are built based on more than 2 tables and 2 string columns.

(1) **Performance on our workloads.** Figure 6.8 (a), (b), and (c) presents the

percentage of queries completed versus the runtime of different methods on QS-base, QS-multi-subs, and QS-diff-wilds, respectively. Compared to the eight baselines, ArbiLIKE achieves runtime reductions of up to 1.52×, 1.74×, and 1.67× for QS-base, QS-multi-subs, and QS-diff-wilds workloads, respectively. Compared to Astrid, ArbiLIKE's runtime is shorter by factors of 1.29×, 1.25×, and 1.41× across these same workloads. Furthermore, ArbiLIKE's runtime is 1.34×, 1.46×, and 1.52× shorter than those of Postgres for QS-base, QS-multi-subs, and QS-diff-wilds, respectively. Postgres shows robustness for complex queries (QS-multi-subs and QS-diff-wilds) due to its heuristic methods that effectively bound estimation errors for intricate queries, with its performance on QS-diff-wilds being only slightly inferior to ArbiLIKE. MO, however, exhibits the longest execution times across the three workloads due to having the largest cardinality estimation errors.

(2) **Performance on JOB workload.** Figure 6.8(d) presents the percentage of queries completed versus the runtime of different methods on the JOB. ArbiLIKE, along with Astrid, S-SPH, Postgres and DREAM, demonstrates superior performance in executing the JOB. This is attributed to their effective methods, such as predicates encoding method, fine grained positional string pattern and heuristics, which more accurately model string patterns. ArbiLIKE completes all JOB queries in approximately 164 seconds, marking a 79.8% reduction in total runtime compared to the slowest method (MO at 294 seconds). Additionally, in comparison with Astrid, P-SPH, Postgres, and DREAM, ArbiLIKE shows execution time reductions of 1.23×, 1.31×, 1.37× and 1.48×, respectively.

In Figure 6.9, we compare the runtime relative to Postgres across 80 queries featuring LIKE predicates from JOB. The term "old_time" refers to Postgres's runtime, while "new_time" denotes the runtime after incorporating ArbiLIKE's cardinality estimations for the LIKE predicates. Out of the 80 queries, 67 exhibit improved runtime, with five queries showing more than 10× relative improvement. For example, query 13B, originally joining "company_names" and "movie_companies" yielding over a million rows, is restructured in the updated plan to join "title" and "company_names" first, reducing the interim result size to under 1000 rows and enabling a more cost-effective subsequent join. Conversely, 13 queries experienced slower runtime, with two notably slower.

(3) **Performance on JOB-extend workload.** Figures 6.10 and 6.11 depict the

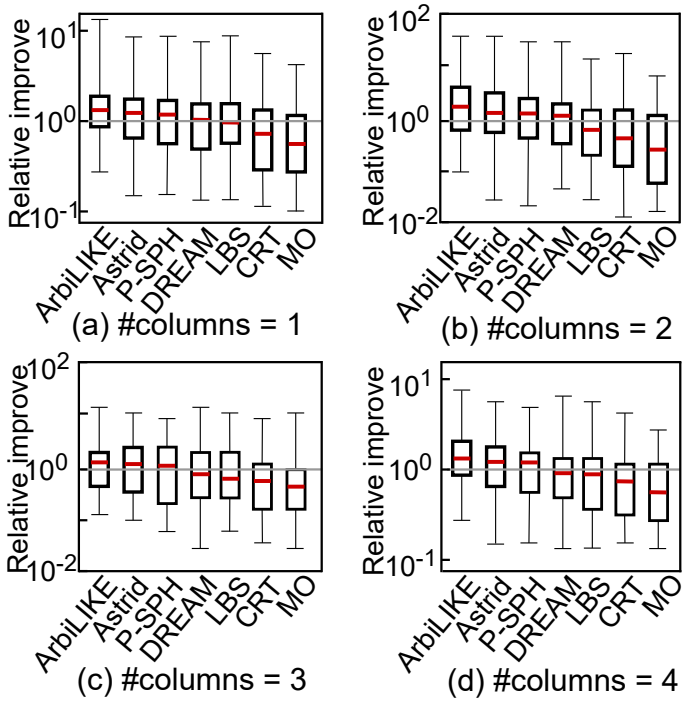


**Figure 6.10:** Relative runtime improvement for queries with LIKE predicates in JOB-extend, where each LIKE predicate is constructed using one or multiple string columns from the same table.

relative runtime improvements by various methods for LIKE predicates derived from differing numbers of tables and columns, with Postgres serving as the baseline for comparison. Methods such as LBS, CRT, BayesNet, and MO show declining performance as the complexity from an increased number of tables and columns in LIKE predicates. The complexity of string patterns within multi-column predicates poses significant challenges for these models to provide accurate pattern matching. In contrast, ArbiLIKE shows exceptional performance, even when managing queries involving up to four tables and four columns, as demonstrated in Figure 6.11(d). This is attributed to ArbiLIKE's advanced predicates encoding and sequence model, which adeptly captures complex string patterns. Such capabilities ensure that ArbiLIKE consistently delivers stable runtime performance, regardless of the query's intricacy. Notably, ArbiLIKE achieves up to a 1.6× reduction in runtime compared to Postgres, this result is achieved when the LIKE predicates based on four tables and four columns.

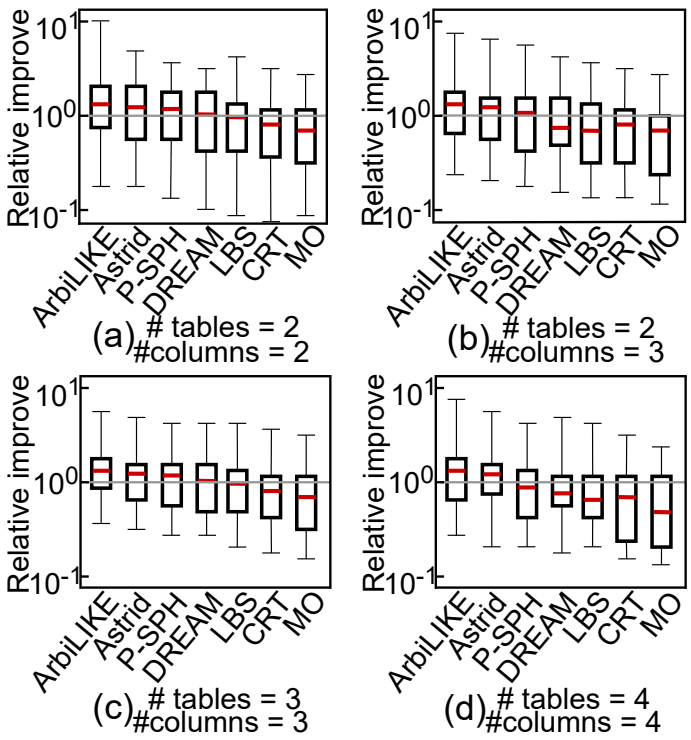


**Figure 6.11:** Relative runtime improvement for queries with LIKE predicates in JOB-extend, where each LIKE predicate is constructed using two or more string columns.

### 6.6.5 Hyper-parameter Tuning

We improve ArbiLIKE's accuracy by tuning its hyper-parameters. To tune hyperparameters for ArbiLIKE, we primarily employed three approaches: (1) Search-based methods; (2) Bayesian optimization (BO); and 3) reinforcement learning (RL). BO-based methods require a well-defined objective function for optimization, while RL-based methods necessitate defining actions, states, and reward functions. Due to the extensive training time required for BO-based and RL-based methods, we opted for a robust and straightforward search-based method.

(1) **Reasons for different hyperparameter values.**

Model architecture. ArbiLIKE consists of three models: $\mathcal{M}_{\text{cent}}$ (§6.3.3), $\mathcal{M}_{\text{sub}}$ (§6.3.4), and $\mathcal{M}_{\text{est}}$ (§6.4.2). In its deep learning models, a balance between complexity and overfitting is maintained. Based on existing studies [30, 113], a sequence model with four layers is efficient for learning underlying string patterns. Thus, ArbiLIKE employs two to five layers, with each layer having 16 to 512 neurons.

Other five hyperparameters. (1) $q$-grams length: Short $q$-grams capture granular pat-

**Table 6.5:** Hyper-parameters considered for tuning.

| Hyper-parameter | Design Space | Chosen Value |
|---|---|---|
| No. layers($\mathcal{M}_{\text{cent}}$) | {2,3,4,5} | 3 |
| No. neurons($\mathcal{M}_{\text{cent}}$) | 16-512 | L1:32, L2:96, L3:48 |
| No. layers($\mathcal{M}_{\text{sub}}$) | {2,3,4,5} | 3 |
| No. neurons($\mathcal{M}_{\text{sub}}$) | 16-512 | L1:48, L2:128, L3:48 |
| No. layers($\mathcal{M}_{\text{est}}$) | {1,2,3,4,5} | 2 |
| No. neurons($\mathcal{M}_{\text{est}}$) | 16-512 | L1:48, L2:256 |
| Length. $q$-grams ($q$) | {1,2,3,4,5} | 3 |
| Subs embedding size ($\lambda$) | 16-80 | 48 |
| Batch size ($B$) | 32-256 | 128 |
| Learning rate ($\alpha$) | $1e^{-5}$-$1e^{0}$ | $5e^{-4}$ |
| Size. signature vector ($L$) | 16-128 | 60 |

terns, while longer ones detect complex patterns. However, longer $q$-grams also raise computational complexity. A typical length of $q$-grams is between two and seven. (2) Substrings embedding size $\lambda$ (§6.3.4): Larger embeddings can lead to overfitting, especially in smaller datasets. Therefore, $\lambda$ is tuned within the range of 16 to 80. (3) Batch size $B$ (§6.4.2): It influences the training dynamics. Smaller batches might converge faster but are less stable, whereas larger batches provide stability at a computational cost. The chosen range for $B$ is 32 to 256. (4) Learning rate $\alpha$: It dictates model adjustment in response to error. A smaller value (around $1e^{-4}$) is typical, but ArbiLIKE explores values from $1e^{-5}$ to $1e^{0}$. (5) Signature Vectors Length $L$ (§6.5.2): Affects the accuracy of approximations. A larger $L$ increases accuracy but also computational load, with the value ranging from 16 to 128.

(2) **How to tune hyperparameters.** (1) Define the hyperparameter grid: Define a grid of hyperparameter values mentioned above. For example, learning rates of 0.001, 0.01, 0.1, etc. (2) Cross-validation setup: Use strategies like k-fold cross-validation for evaluation. From our collection of datasets (Table 6.2 and string columns in IMDB), we randomly choose one for testing and use the remaining for validation. (3) Search for hyperparameter values: Use uniform grid search algorithm [119] to evaluate each combination of five hyperparameter values in the grid. For model structures tuning, ArbiLIKE uses the Optuna [103] which is designed to identify optimal neural network structures. We then test ArbiLIKE with each of these configurations against workloads (Table 6.3, JOB, and JOB-extend) from the testing and validation datasets. (4) Finally, we select the winning model structures and hyperparameter configuration. The hyperparameter tuning process is integral to optimizing ArbiLIKE's accuracy,

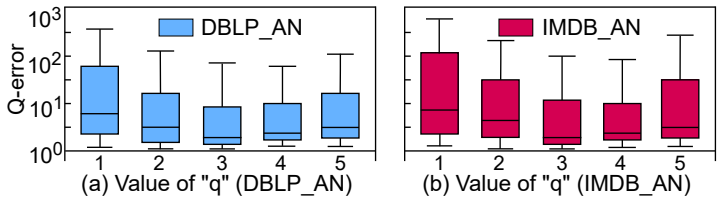


**Figure 6.12:** The impact of length of $q$-grams in ArbiLIKE (x-axis) on Q-errors (y-axis).

**Table 6.6:** This table shows the impact of substring embedding size on the estimation results (§6.3.4). The result shows the 90th error across five datasets on QS-base workload. The lowest errors are bolded.

| Embed Size | DBLP_AT | DBLP_AN | IMDB_MN | IMDB_MT | IMDB_AN |
|---|---|---|---|---|---|
| 16 | 19.2 | 50.3 | 74.1 | 98.3 | 142.9 |
| 32 | 15.4 | 23.4 | 55.6 | 37.3 | 46.8 |
| 48 | **9.2** | 9.8 | **14.2** | 10.8 | **12.7** |
| 64 | 10.3 | **9.5** | 14.5 | 10.1 | 13.4 |
| 80 | 9.7 | 10.3 | 14.7 | **9.6** | 13.8 |

ensuring that it is well-suited to the specificities of the datasets and workloads it encounters. Table 6.5 provides details of our proposed model structures and five hyper-parameter values in ArbiLIKE after the hyperparameters tuning.

(3) **Impact of hyperparameters tuning** (1) $q$-gram Length. Figure 6.12 shows that the optimal $q$-gram length for ArbiLIKE is $q = 3$. Short $q$-grams ($q = 1$) inadequately represent database strings, while longer ones ($q \geq 5$) create substrings that deviate from LIKE predicate strings, both resulting in higher estimation errors. (2) Substring Embedding Size $\lambda$. A larger $\lambda$ helps ArbiLIKE learn complex substring correlations, but very large values risk overfitting. Table 6.6 presents the experimental results as we adjust the embedding size between 16 and 80. Experiments reveal that $\lambda$ values beyond 48 see diminishing error reduction, guiding the selection of $\lambda$ in ArbiLIKE's tuning process. (3) Signature Vector Size $L$. Signature vector size impacts set resemblance accuracy in cardinality estimations. Figure 6.13(a) shows the influence of signature vector size, $L$, on the Q-errors for the QS-multi-subs workload across the five datasets (Table 6.2). Increasing the size up to 50 significantly reduces Q-errors, but sizes above 80 may introduce inaccuracies. Thus, ArbiLIKE uses a signature vector size of 60.

(3) Size of signature vectors $L$. The signature vector plays a crucial role in calculating

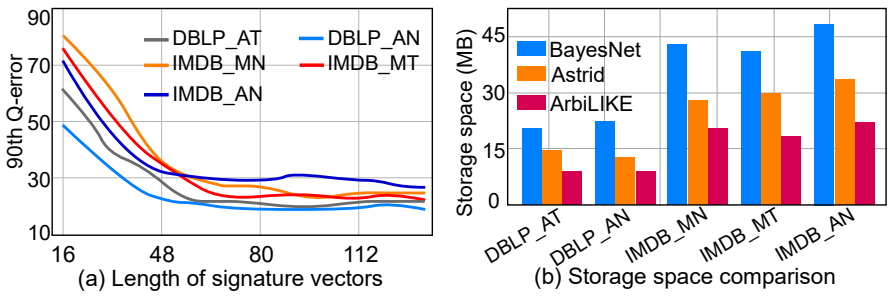


**Figure 6.13:** Signature vector size and storage space of ArbiLIKE.

set resemblance when estimating cardinalities for generic LIKE predicates (§6.5.2). Figure 6.13(a) shows the influence of signature vector size, $L$, on the Q-errors for the QS-multi-subs workload across the five datasets (Table 6.2). There's a pronounced reduction in Q-error as the size of the signature vectors increases (specifically for sizes less than 50). In ArbiLIKE, we utilize hash functions to find appropriate values for elements in signature vectors, ensuring that similar elements consistently hash into identical buckets. However, a large signature vector size (beyond 80) can occasionally introduce false positives or even false negatives, thereby compromising the accuracy of the set resemblance computation. In ArbiLIKE, we select 60 as the size for each signature vector.

### 6.6.6 Efficiency of ArbiLIKE

**Training time comparison.** Figure 6.14(a) compares training time among various models across different datasets. DREAM's training spans between 192 and 466 mins across all datasets, with the bulk of this time dedicated to the constructing trie-based dynamic programming table. Astrid's training time varies from 150 to 244 mins, with the majority of this time attributed to the embedding learning process. Because Astrid's embedding method depends on large prefix and suffix trees, which can even exceed the size of DB. ArbiLIKE is the most time-efficient in training. This is attributed to two main factors. First, ArbiLIKE's superior efficiency in LIKE predicate embeddings over Astrid: its substring embedding method is bounded by the number of fixed-length ($q$=3) substrings in the DB. Secondly, ArbiLIKE uses a light-weighted sequence model as the estimator. Consequently, ArbiLIKE's training time ranges only from 56 to 127 mins.

**Inference time comparison.** Figure 6.14(b) compares the inference time of BayesNet,

DREAM, Astrid, and ArbiLIKE on the QS-base workload. Among the leaning-based estimators, BayesNet has the shortest inference time, because it just relies on probability multiplications for different substrings to estimate cardinalities, which is lightweight. DREAM is faster than Astrid and ArbiLIKE, because it only involves finding the minimal string in the dynamic programming table and feeds the feature into a sequence model for inference. The sequence models in Astrid needs more floating-point operations during the inference compared to ArbiLIKE, leading to its inference time spans from 20 ms to 176 ms. On average, Astrid's median inference time is 2.2× longer than that of ArbiLIKE. The inference time of leaning-based estimators are closely tied to the string length in LIKE predicates, with short strings (less than 8 characters) requiring 20-80 ms and longer strings (over 20 characters) requiring over 150 ms. While ArbiLIKE's estimator has only 176 inference neurons (48+128) with roughly 8448 weights. As a result, ArbiLIKE only requires 8448 Multiply-Accumulate operations per inference.

**Storage space.** ArbiLIKE's storage needs primarily stems from three components: substring embeddings $\mathbf{E}_{\text{sub}} \in \mathbf{R}^{N_{sub} \times \lambda}$, model parameters ($\mathcal{M}_{\text{cent}}$, $\mathcal{M}_{\text{sub}}$, and $\mathcal{M}_{\text{est}}$), and signature vectors $\mathbf{Sig} \in \mathbf{R}^{N_{sub} \times L}$, where $N_{sub}$ is the number of substrings extracted from the DB, $\lambda$ is the embedding size, and $L$ is each signature vector's size. The total data storage for ArbiLIKE is approximately $108 \times N_{sub} + 454,656$. Thus, the storage directly correlates with the number of substrings $N_{sub}$. For instance, with 47,891 substrings from the IMDB_AN dataset, ArbiLIKE's storage space is around 21.5MB using a single-precision floating-point for the data. Figure 6.13(b) compares the storage demands of BayesNet, Astrid, and ArbiLIKE. ArbiLIKE uses 2.3×-3.1× less storage space than its counterparts. BayesNet's storage space stems from the size of its Conditional Probability Tables (CPTs) and its network structure. In our tests, the size of BayesNet's CPTs ranges from 22.5MB to 49.4MB, making BayesNet consumes the most storage space. Meanwhile, Astrid retains prefix and suffix trees derived from the DB along with the parameters of its trained estimator. Astrid's storage needs range between 13.8MB and 33.5MB, making it consume 1.3×-1.7× more storage than ArbiLIKE.

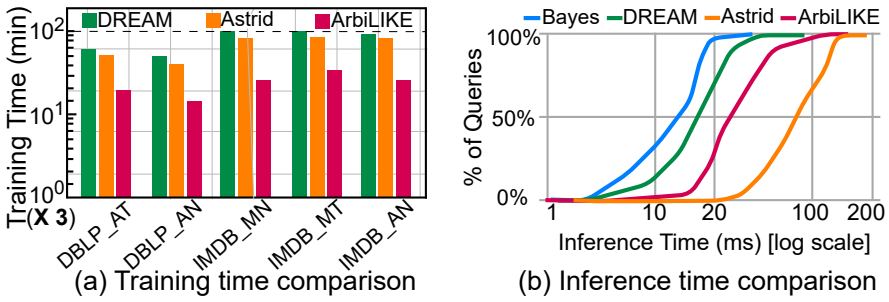


**Figure 6.14:** Physical efficiency of ArbiLIKE.

### 6.6.7 Handling String Indexing

We explore how ArbiLIKE can be leveraged to deal with the problem of string indexing. The output from string indexing indicates the count of strings in a dataset that are lexicographically less than or equal to a specified query string $Q_s$, with "s" representing the string within the query.

**Dataset.** For our evaluation, we use five datasets: "Article_titles", "Movie_notes", "Movie_titles", "URLs", and "GEO". The details of "Article_titles", "Movie_notes", and "Movie_title" are listed in Table 6.2. The "URLs" [120] dataset, comprising 92 million quoted URLs with an average string length of 20 characters, and "GEO", containing 7 million global geographic location names with an average string length of 13 characters, are both drawn from real-world data.

**Baselines.** We use SIndex [121], RSS [122], Masstree [123], and Wormhole [124] as baselines for string indexing. SIndex and RMI are modern string indexing methods based on learned models. Masstree combines a B-tree with a Trie for concurrent indexing, while Wormhole integrates a Trie into the internal nodes of a B-tree. These methods are implemented in C++ or C for efficiency, whereas ArbiLIKE is developed in Python. Consequently, we assess the string indexing error offline, reporting the *Mean Absolute Error* (MAE [125]) for each method.

**Configurations.** In contrast to typical machine learning tasks where overfitting is avoided, ArbiLIKE is designed to fit the input dataset as closely as possible for precise indexing, hence we utilize the entirety of each dataset for both training and evaluation. For strings longer than the $q$-gram length, we employ ArbiLIKE's encoding method (§6.3) to convert distinct strings into uniform-length real-valued vectors. These vectors are then processed by the sequence model (§6.4), outputting a value

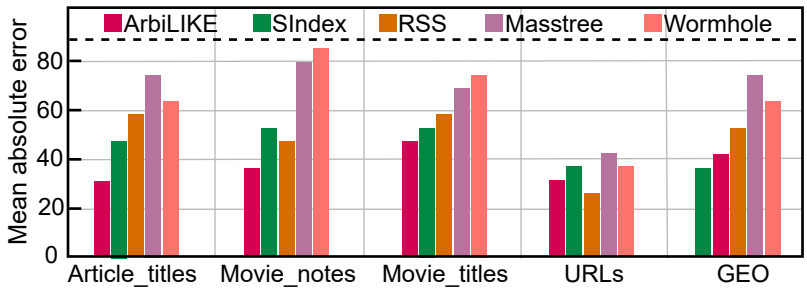


**Figure 6.15:** Mean absolute errors for the string indexing of different methods. x-axis is the name of each datasets, y-axis is the indexing error.

between 0 and 1, which is subsequently scaled to an integer range.

**Results.** We randomly select 2,000 strings from each dataset for indexing. Figure 6.15 shows the indexing error in terms of MAE. ArbiLIKE's MAE is up to 1.4× lower than the baselines on "Movie_notes", "Movie_title", and "GEO" datasets. On the "URLs" dataset, however, ArbiLIKE's MAE is worse than the other two learned models (SIndex and RMI), due to "URLs" has an extensive number of entries, which poses a challenge for single learned models predicting across a wide range. Compared to ArbiLIKE, both SIndex and RMI contain a hierarchical model to improve the indexing accuracy. Nonetheless, ArbiLIKE's MAE is still 1.2× lower than the Trie and B-tree based models (Masstree and Wormhole), indicating the benefits of learned models over the traditional string indexing techniques. However, we find that ArbiLIKE requires some hours to be trained for the string indexing problem, while the construction time of Masstree and Wormhole only take just a few minutes.

### 6.6.8 Handling Data Updates

We analyze how ArbiLIKE performs in a dynamic environment.

**Dynamic environment setup.** Suppose that there are $n$ queries uniformly distributed in a time range $[T_i, T_{i+1}]$, and $T = T_{i+1} - T_i$ which controls the frequency of the data update. The queries based on updated data begin to come at timestamp $T_i$. Suppose the model update finishes at timestamp $T_f$. If $T_f \in [T_i, T_{i+1}]$, for thefirst $\lceil n \cdot \frac{T_f - Ti}{T} \rceil$ queries, their cardinalities are estimated using the stale model. For the remaining $\lfloor n \cdot (1 - \frac{T_f - Ti}{T}) \rfloor$ queries, the updated model will be used. **Data update.** We use IMDB_AN dataset (Table 6.2) for testing under a dynamic environment, this is the string attribute "Actor Names" in table `title` from IMDB [23]. We use the

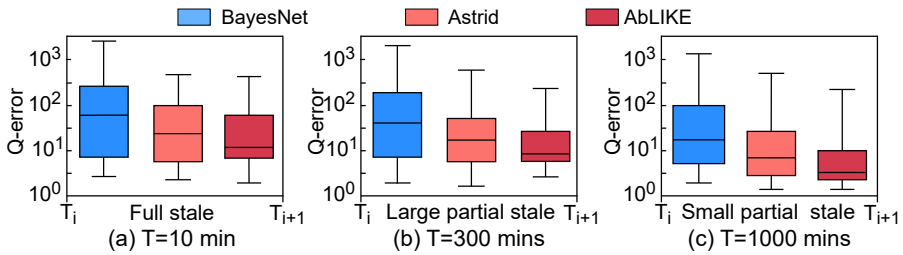


**Figure 6.16:** Estimation quality under dynamic environment.

similar method introduced in [65, 19] to update the dataset. In particular, we partition the table `title` into two parts on the `year` column. The part with the latest `year` is used as the new data to be appended into dataset. After data updates, we apply our workload generation method (§6.6.1) on the updated dataset to generate 5K queries for testing. These queries are uniformly distributed in $[T_i, T_{i+1}]$. $T$, which is equal to $T_{i+1} - T_i$, is a parameter, which represents how "frequently" the data are updated. For example, if the data are periodically updated every 100 mins, then $T$ is 100 mins. **Model update.** We apply incremental learning [126] to update BayesNet, Astrid, and ArbiLIKE, subsequently comparing their estimation accuracy. Since all three are data-driven estimators, they need to be retrained on the entire new updated dataset.

**Estimations in a dynamic environment.** We test the estimation quality of ArbiLIKE, BayesNet, and Astrid in a dynamic environment. We manipulate the value of $T$ to adjust the data update frequency, setting it at high (10 min), medium (300 mins), or low (1000 mins). Figure6.16 illustrates the estimation quality of BayesNet, Astrid, and ArbiLIKE in the dynamic environment.

With a high update frequency ($T$=10 mins), the estimators ArbiLIKE, BayesNet, and Astrid cannot complete their model updates in time. As a result, all queries within the $[T_i, T_{i+1}]$ interval are processed using the stale model, leading to inaccurate estimations for the three models (Figure 6.16(a)). With a medium data update frequency ($T$=300 mins), all three learned estimators-ArbiLIKE, BayesNet, and Astrid-successfully complete their model updates. However, in this scenario, a large portion of queries coming within the time interval $T$ are still evaluted using stale models. As depicted in Figure 6.16(b), ArbiLIKE's accuracy is up to 10.4× and 4.6× better than BayesNet and Astrid, respectively. With a low data update frequency ($T$=1000 mins), all three estimators can still finish model updates within time inter-

**Table 6.7:** This table shows the impact of different encoding methods for LIKE predicates(§6.3.4). "PST" represents the encoding method based on prefix and suffix trees in Astrid. "Ours" denotes our encoding method for substrings. The lowest errors are bolded.

| Query Set | Encoding | 50th | 90th | 95th | 99th |
|---|---|---|---|---|---|
| QS-base | Binary | 5.77 | 65.7 | 365.6 | 1206.8 |
| | PST | 3.56 | 18.5 | 89.6 | 186.5 |
| | **Ours** | **1.86** | **6.7** | **12.7** | **38.1** |
| QS-Multi-Subs | Binary | 5.67 | 127.4 | 872.1 | 1462.9 |
| | PST | 4.17 | 42.8 | 208.6 | 985.3 |
| | **Ours** | **2.35** | **14.6** | **23.3** | **117.9** |
| QS-Multi-Wildcards | Binary | 6.02 | 102.4 | 789.7 | 1562.6 |
| | PST | 4.63 | 47.7 | 230.6 | 1146.8 |
| | **Ours** | **2.55** | **17.4** | **26.0** | **227.1** |

val $T$. Here, only a minor fraction of queries rely on stale models. Consequently, as Figure 6.16(c) shows, the accuracy of all three models improves when compared to their accuracy under high and medium data update frequencies.

Assume the time of finishing model update for ArbiLIKE, BayesNet, and Astrid are $T_{arbi}$, $T_{bayes}$, and $T_{astr}$ respectively, therefore, the number of queries coming within the time interval $T = T_{i+1} - T_i$ that are tested by stale models for ArbiLIKE, BayesNet, and Astrid are $n_{arbi} = \lceil n \cdot \frac{T_{arbi}-Ti}{T} \rceil$, $n_{bayes} = \lceil n \cdot \frac{T_{bayes}-Ti}{T} \rceil$, and $n_{astr} = \lceil n \cdot \frac{T_{astr}-Ti}{T} \rceil$ respectively. Here, $n$ is the total number of queries coming within time interval $T$. According to the efficiency of ArbiLIKE (§**??**), we have $T_i < T_{arbi} < T_{bayes} < T_{astr} < T_{i+1}$, therefore, we can conclude $n_{arbi} < n_{bayes} < n_{astr}$. This means compare to the updated BayesNet and Astrid, more queries will be tested by the new updated ArbiLIKE. As a result, ArbiLIKE performs better than BayesNet and Astrid in a dynamic environment.

### 6.6.9 Impact of Embedding Methods

We study how the embedding methods (§6.3.4) impacts accuracy. **Impact of embedding methods.** We encode the substrings of string tuples in DB into feature vectors using our embedding method. Existing estimators [16, 66, 30] often use binary or prefix and suffix tree (PST) embedding. Table 6.7 shows the impact of different embedding methods on errors across three LIKE predicates query sets (§6.6.1) for the IMDB_AN dataset. While the impact on low-quantile errors is small, different

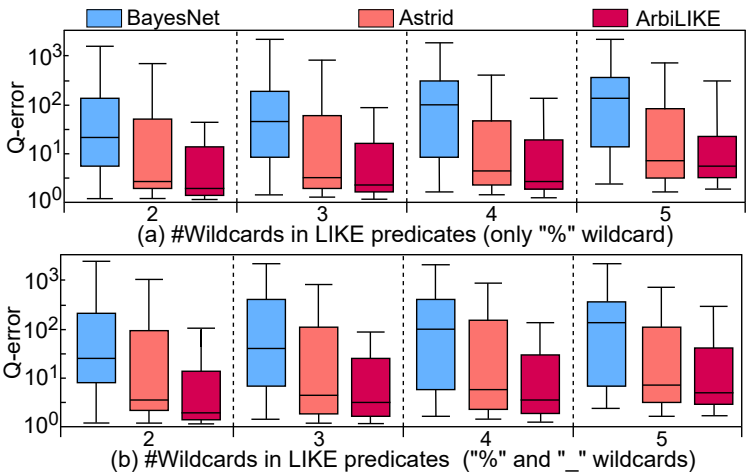


**Figure 6.17:** The impact of different number of wildcards (x-axis) on Q-errors for LIKE predicates (y-axis).

embedding methods significantly affect high-quantile errors. Compared to PST-based embedding, our embedding method achieves accuracy gains up to 8.4× on high-quantile errors across different query sets. This improvement results from ArbiLIKE's multi-tiered contrastive embedding approach.

We transform the substrings of database string tuples into feature vectors using our embedding method. Existing estimators often use binary or prefix and suffix tree (PST) embedding. Table 5 illustrates the effect of different embedding methods on errors across three LIKE predicates query sets for the IMDB_AN dataset. While the impact on low-quantile errors is minimal, different embedding methods significantly affect high-quantile errors. Our method reduces the 99th percentile error notably, compared to binary and PST-based embeddings, achieving accuracy gains up to 4.9×, 8.4×, and 5.1× on high-quantile errors across different query sets. This improvement results from ArbiLike's multi-tiered contrastive embedding approach.

### 6.6.10 Ablation Study of ArbiLIKE

We evaluate the relative importance of three components of ArbiLIKE. Table 6.8 shows the results. The base estimator utilizes a predicate encoding method grounded in the prefix and suffix trees (PST), as seen in Astrid [30]. In variant (A), we adopt ArbiLIKE's LIKE predicate encoding method (§6.3), replacing the PST used in the base estimator. The encoding method in ArbiLIKE notably enhances estimation ac-

**Table 6.8:** Ablation studies: varying primary components of ArbiLIKE. We show the impact of (A) LIKE predicates encoding method (§6.3), (B)learned estimator (§6.4), and (C) set resemblance for arbitrary LIKE predicates (§6.5) on QS-multi-subs and QS-diff-wilds workloads over IMDB_AN dataset.

| **Methods** | **QS-multi-subs** | | | **QS-diff-wilds** | | |
|---|---|---|---|---|---|---|
| | **p50** | **p90** | **p99** | **p50** | **p90** | **p99** |
| Base estimator | 11.3 | 210.5 | 3030.1 | 14.9 | 341.4 | 3624.5 |
| (A)+ Predicates encoding | 6.8 | 72.3 | 891.2 | 7.4 | 106.7 | 1159.6 |
| (B)+ Learned estimator | 3.9 | 53.3 | 424.4 | 4.1 | 71.2 | 642.5 |
| (C)+ Set resemblance | 2.3 | 21.4 | 117.9 | 2.6 | 25.4 | 217.3 |

curacy, especially in high-quantile errors. Specifically, by incorporating our encoding method, we observe reductions in the 50th, 90th, and 99th percentile errors by up to factors of 2.0×, 3.2×, and 3.4× respectively, compared to the base estimator. In variant (B), we enhance variant (A) by employing our importance-boosted sequence model (§6.4) to estimate the cardinality for LIKE predicate, instead of using the LSTM in base estimator. Our sequence model significantly improves both low and high-quantile errors. Specifically, reductions in the 50th, 90th, and 95th percentile errors are observed by factors of up to 1.8×, 1.5×, and 2.1×, respectively, when compared to the variant (A). In variant (C), we enhance variant (B) by integrating our set resemblance method (§6.5) to estimate cardinalities for generic LIKE predicates, replacing the technique from CRT. Our set resemblance approach markedly reduces errors across all quantiles when compared to the variant (B). Notably, the error reduction at the 50th, 90th, and 99th percentiles reach up to factors of 1.7×, 2.8×, and 3.6× respectively. Among the three components, our set resemblance method significantly reduces high-quantile errors.

### 6.6.11 Varying Number of Wildcards

We evaluate three learning-based approaches — BayesNet, Astrid, and ArbiLIKE - when handling LIKE predicates with varying numbers of wildcards (ranging from two to five). Figure 6.17 illustrates the Q-errors for QS-multi-subs and QS-diff-wilds workloads on the IMDB_AN dataset, and each workload contains 2000 LIKE predicates. The predicates are categorized into four groups based on the count of wildcards (from two to five). ArbiLIKE demonstrates lower errors than both BayesNet and Astrid, irrespective of the number of wildcards present in the LIKE predicates.

Notably, the estimation accuracy of ArbiLIKE remains fairly stable and is not significantly affected by the wildcard count in LIKE predicates. For instance, when the number of wildcards rises from two to five, the 75th percentile error of ArbiLIKE merely ascends from 16.3 to 18.8 and from 15.4 to 21.3 on the QS-multi-subs and QS-diff-wilds workloads respectively. The consistency in ArbiLIKE's estimation error is attributed to our methodology for estimating cardinalities for arbitrary LIKE predicates (§6.5.1).

## 6.7 Summary

The cardinality estimation using machine learning models is a new research trend in the database community. However, it is challenging to make accurate cardinality estimations using machine learning models for LIKE predicates. We introduce ArbiLIKE to address this problem. ArbiLIKE has a new LIKE predicates embedding method which make the input feature vectors more informative for the cardinality estimation. It also includes a substring-importance enhanced sequence model as a cardinality estimator for LIKE predicates. ArbiLIKE can handle generic LIKE predicates with wildcards ("%", "_"). Experimental results show that ArbiLIKE has 1.46-94.6× higher accuracy than state-of-the-art solutions.

# Chapter 7

# Related Work

The performance optimization for data-intensive applications like auto-labeling and cardinality estimation is challenging. T here are some proposed approaches aimed to tackle the challenges of data labeling and cardinality estimation in database.

**Automatic labeling.** We first provide an overview of automatic labeling methods, which label data automatically based on generated labeling functions using both labeled and unlabeled data.

The main challenge of auto-labeling is to build proper labeling functions that can cover the most data instances in the dataset [127, 128, 129, 130, 131, 132, 133, 134, 135]. Labeling functions with high quality are difficult to be acquired. In [10], Varma et. al propose a method that uses machine learning models to build labeling functions under weak supervision. Other work [136, 137, 9, 138] uses distant supervision [136, 137, 139], in which the training sets are generated with the help of external resources, such as knowledge bases. Some recently proposed approaches [140, 9] demonstrate the use of proper strategies to boost the labeling quality by ensemble technique [133, 141]. The existing approaches focus on static datasets with fixed size and pre-determined number of labels. Our work focuses on the dynamically increased datasets on mobile devices. Our work has the capability to identify new labels that are never seen before. To solve the hardware resource constraint problem on mobile devices, we leverage processor heterogeneity to efficiently run the auto-labeling workload. Our work not only labels dataset with high quality, but also is highly feasible to be deployed on mobile devices.

**Optimization of machine learning on mobile devices.** Recently, there are

many existing efforts that optimize the performance of machine learning models on both server side [142] and edge side, including dynamic resource scheduling [143, 144, 145, 146, 147, 148, 149, 150, 151, 152], computation pruning [153, 154, 155, 156, 157], model partitioning [158, 149, 159, 160], model compression [161, 162, 163], coordination with cloud servers [164, 158] and memory management [4, 148]. Flame is different from them, because it focuses on data labeling task on heterogeneous mobile processors. In particular, DeepX [149] proposes a number of resource scheduling algorithms to decompose DNNs into different sub-tasks on mobile devices. LEO [164] introduces a power-priority resource scheduler to maximize energy efficiency. NestDNN [161] compresses and prunes models based on the available hardware resource on mobile devices.

**Deep Learning for Databases.** Recently, there has been extensive work on applying techniques from DL for solving challenging database problems. One of the first work was by Kraska et. al [165] that sought to build learned indexes. There are two conceptual connections between our approach and [65]. First, they both seek to leverage the data distribution for the task at hand. Second, both of these leverage the concept of ensembles/mixtures to improve the performance of individual DL models. However, the problem they tackle are very different – density estimation vs indexing. Another difference is that [65] uses the cumulative distribution function (CDF) while our unsupervised approach uses probability density estimation (PDF). There has been extensive work on using DL techniques including reinforcement learning for query optimization (and join order enumeration) such as [16, 17, 19, **?**]. Recently, there has been effort to build a learned database systems [166] and an end-to-end learned optimizers [18]. DL has also been applied to the problem of entity resolution in and data integration [167].

**Query-driven cardinality estimators.** By leveraging past or collected queries, query-driven approaches build and correct the current models to learn functions mapping a query with its predicted probability. Representative work includes correcting histograms [75, 168], updating statistical summaries in DBMS [169, 170], and query-driven kernel-based methods [171, 172].

**Data-driven cardinality estimators.** Data-driven approaches build unsupervised models, which learn the joint probability density function (PDF) of table attributes to estimate the probability of an query. In recent researches, there has been

extensive work on applying data-driven techniques for solving challenging database problems. *Sample and Kernel-based methods* [17, 21, 21, 171] sample records from tables on-the-fly, or use average kernels centered around sampled points for estimation. *Sum-Product Networks (SPNs)* [173, 174] estimate the PDF results using either sum and product operations to combine children information in a tree structure. *Deep Auto-Regression (DAR)* models are the current state-of-the-art density models [175, 166, 176, 177, 18] from the ML community. DAR models capture all possible correlations among the attributes of tables to produce selectivity estimates.

**ML based approaches for selectivity estimation.** Recently, there has been a surge of interest in using ML-based methods in order to enhance the performance of database components, e.g. indexing [178], data layout [179], query execution [167] and scheduling [180]. Most recently, work [16] targeted correlations across joins using a custom neural network architecture. While [16] certainly addresses generic version of the selectivity estimation problem, the models in this paper are much more succinct leading to significantly faster estimations.

**Optimizing DNN training time.** Solutions proposed for reducing DNN training times include specialized hardware [181, 182], parallel training [183, 184, 185], GPU memory optimizations [186], lowering communication overhead [187], and operator optimizations [188, 189, 190, 191]. New hardware systems like NVIDIA's Magnum IO [181, 192] provide high-throughput storage solutions to reduce data loading overheads, but cannot help when training DNNs in distributed environments with complex storage hierarchies. Model batching [193] addresses data loading costs when running multiple DNNs on a single node. OneAccess [194, 195, 196] is a preliminary study that makes a strong case for storing pre-processed data across epochs to reduce the data preprocessing overhead. However such an approach precludes commonly used online data preprocessing techniques, which can affect model convergence during training. None of these approaches address, as does Lobster, data loading overheads resulting from load imbalances across GPUs. Lobster balances loads between GPUs and avoids bursty data loading such that the data loading does not become a performance bottleneck.

**Data caching for distributed DNN training.** Quiver [36, 197] uses SSD as caches to avoid slow data loading from remote storage. Cerebro [198] partitions the dataset across nodes in a cluster. Instead of shuffling data, Cerebro moves the models

from one node to others. However, when training DNNs with a single node, using Cerebro cannot bring performance improvement. DeepIO [199] uses a partitioned caching technique for distributed training. DeepIO heavily relies RDMA for high performance I/O. DIESEL [200] deploys a task-grained distributed cache across nodes for multiple DNN training tasks. DIESEL introduces metadata snapshot mechanisms for each training dataset, and mainly focuses on optimizing metadata processing during the DNN training. MinIO [35] reduces the amount of disk I/O for training on a single node and multiple nodes. MinIO does not provide the fine-grained cache strategy like Lobster. For MinIO, once data samples are cached, they are never evicted out of the cache. NoPFS [22] introduces a performance model that can leverage the storage hierarchy for the data caching. But NoPFS cannot immediately evict data samples with long reuse distances out of memory. Lobster addresses the I/O bottleneck by providing the thread management for different stages in the training pipeline. Furthermore, Lobster leverages the knowledge on the reuse distance of data samples to make the best use of the memory cache. In particular, we can evict "cold" data samples out of the memory cache to save the memory cache space and prefetch more data samples to be accessed in the near future into the memory cache.

**Cardinality estimation for string queries.** The early work [25, 26, 27, 28, 29, 108, 201, 107, 202, 203] addressed cardinality estimation for LIKE predicates based on suffix trees, built from the substrings of string tuples in a DB. An improved estimator was given by [29, 108]. Recent studies, such as CRT [202], improved estimates by identifying shorter strings with cardinalities similar to a given string. A followup work [26] introduced two estimators, HSol and Vsol. A considerable amount of works [25, 27] have been done in the field of approximate string cardinality estimation. Other techniques for numeric data include sampling [204, 25, 27], histograms [107], wavelets [205], and graphical models [179].

Early studies addressed LIKE predicates cardinality estimation using suffix trees, built from database string tuples. Despite improvements in estimation by subsequent works, underestimation persisted. Recent studies, such as CRT, improved estimates by identifying shorter, similar cardinality strings. Follow-up research introduced HSol and Vsol estimators, leveraging set hashing for estimation. A significant volume of research, including [8] and [13], has been conducted on approximate string cardinality estimation. Other techniques for numeric data encompass sampling [18], histograms

[23], wavelets [17], kernel density estimation [19], and graphical models.

Various DL-based approaches have been used for cardinality estimation [16, 31, 17, 18, 19, 78, 20, 206, 207, 112, 113, 30, 65, 208, 209, 210, 211]. A recent work [19] used auto-regressive models for cardinality estimation. An effort [31] focuses on cardinality estimation with uncertainties. An empirical analysis of various approaches can be found in [19, 78]. Recent studies have addressed the challenges of more complex query types, such as group-by [206] and range queries [78]. Meanwhile, unsupervised approaches [208] also offer promising estimation results. The use of (deep) learning for approximate query processing, such as in [212], is another promising area of research. In addition, there has been some preliminary work for approximating the edit distance [210] and use it for nearest neighbor search [112]. Unfortunately, there is limited support for string predicate queries.

# Chapter 8

# Conclusion and Future Work

**Conclusions**. We have explored two data-intensive applications including data labeling on mobile devices, cardinality estimation for numeric and non-numeric columns in database system, and data loading for large-scale DNNs training. We also demonstrated how to optimize the performance for these data-intensive applications. In this chapter, we summarize the topics of this dissertation and directions that we plan to study in the future.

This dissertation first presents Flame, the first auto-labeling system for mobile devices, named Flame, to address the above problem. Flame includes auto-labeling algorithms to detect unknown labels from dynamic data; It also includes an execution engine that executes labeling workloads on heterogeneous mobile processors.

Then this dissertation proposes Fauce to address this problem. Fauce has a new query featurization method which can make the input feature vectors more informative for the cardinality estimation. It also includes uncertainty information for estimation results. Experimental results show that Fauce has up to 1.16-91× higher accuracy than state-of-the-art solutions. By leveraging the uncertainty information, Fauce's estimation can be further improved.

This dissertation also investigates the data loading probolem for large-scale DNNs training. Data loading is becoming a major performance bottleneck in distributed DNN training. Prior studies of data loading performance for distributed DNN training have conducted neither a holistic analysis of all training pipeline stages not a fine-grained analysis of the load of individual GPUs, two areas that present opportunities for further optimization. To fill this gap, we have proposed Lobster, a data

loading runtime that exploits several observations related to load imbalance, performance bottlenecks in various stages of the training pipeline, and the reuse distance of training samples to propose a new flexible thread management strategy and cache eviction policy that complements deterministic prefetching. These methods allow Lobster to consistently outperform the state-of-art PyTorch I/O, DALI, and NoPFS systems by 1.3–2.0×. Finally, this dissertation investigates how to make accurate cardinality estimations for LIKE predicates is a significant challenge problem. Our solution, ArbiLIKE, addresses this with a novel embedding method which make the input feature vectors more informative for the cardinality estimation. ArbiLIKE also leverages a substring-importance boosted sequence model and can effectively process generic LIKE predicates with different number of wildcards ("%", "_"). Experiments indicate ArbiLIKE's accuracy to be up to 165.1× greater than leading solutions.
**Future work**. Although we believe that the optimization for data-intensive applications of this dissertation are beneficial and useful, they also have limitations.

- In Chapter 3 proposes runtime solutions to solve the data placement of data labeling workload on the heterogeneous mobile processors. In this work, we assume a single component of Flame occupies a particular kind of computing unit. However, in current mobile processors, it is also common for multiple computing components to share the same computing unit in the mobile processors.

- In Chapter 4, we can explore some advanced embedding techniques for complex queries in our future work. Develop sophisticated embedding techniques that can accurately represent complex query patterns, including those involving LIKE predicates or nested queries. This could involve leveraging advancements in natural language processing (NLP) and graph neural networks (GNNs) to better capture the semantics and structure of queries in a high-dimensional space, enhancing the accuracy of cardinality estimates.

- In Chapter 5, the caching of the training samples in the node-local memory hierarchy suffers from inefficient eviction. This is another point that can be optimized for large-scale DNNs training. Caching is essential for reducing the I/O overheads associated with accessing a remote storage repository and is often implemented in a distributed fashion: each compute node exposes its local

cache to other compute nodes, greatly reducing the need for the compute nodes as a group to interact with the storage repository. By using a pseudo-random number generator to sample the training data, it is possible to obtain foreknowledge of the order in which the training samples will be accessed in future iterations. State-of-the-art approaches leverage such foreknowledge to prefetch training samples in advance, further reducing the I/O overheads. However, prefetching also causes evictions from the cache, which may lead to an undesired situation where the cache is used for samples that will be accessed in the far future, at the expense of those needed in the near future.

# Bibliography


[1] Lucian Codrescu, Willie Anderson, Suresh Venkumanhanti, Mao Zeng, Erich Plondke, Chris Koob, Ajay Ingle, Charles Tabony, and Rick Maule. Hexagon dsp: An architecture optimized for mobile multimedia and communications. *IEEE Micro*, 34(2):34–43, 2014.

[2] Wenqian Dong, Jie Liu, Zhen Xie, and Dong Li. Adaptive Neural Network-Based Approximation to Accelerate Eulerian Fluid Simulation. In *International Conference for High Performance Computing, Performance Measurement, Modeling and Tools (SC)*, 2019.

[3] Wenqian Dong, Zhen Xie, Gokcen Kestor, and Dong Li. Smart-PGSim: Using Neural Network to Accelerate AC-OPF Power Grid Simulation. In *International Conference for High Performance Computing, Performance Measurement, Modeling and Tools* , 2020.

[4] Zhen Xie, Wenqian Dong, Jie Liu, Ivy Peng, Yanbao Ma, and Dong Li. Md-hm: memoization-based molecular dynamics simulations on big memory system. In *Proceedings of the ACM International Conference on Supercomputing*, pages 215–226, 2021.

[5] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[6] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. PipeDream: Generalized pipeline parallelism for DNN training. In *27th ACM Symposium on Operating Systems Principles*, page 1–15, Huntsville, Canada, 2019.

[7] Andrei Ivanov, Nikoli Dryden, Tal Ben-Nun, Shigang Li, and Torsten Hoefler. Data movement is all you need: A case study on optimizing transformers. In *4th Conference on Machine Learning and Systems*, Virtual, 2021.

[8] Ching-Hsiang Chu, Pouya Kousha, Ammar Ahmad Awan, Kawthar Shafie Khorassani, Hari Subramoni, and Dhabaleswar K. Panda. NV-group: Link-efficient reduction for distributed deep learning on modern dense GPU systems. In *34th ACM International Conference on Supercomputing*, pages 1–12, Virtual, 2020.

[9] Alexander Ratner, Stephen H Bach, Henry Ehrenberg, Jason Fries, Sen Wu, and Christopher Ré. Snorkel: Rapid training data creation with weak supervision. *Proceedings of the VLDB Endowment*, 11(3):269–282, 2017.

[10] Paroma Varma and Christopher Ré. Snuba: Automating weak supervision to label training data. *PVLDB*, 12:223–236, 2018.

[11] Yitao Chen, Saman Biookaghazadeh, and Ming Zhao. Exploring the capabilities of mobile devices in supporting deep learning. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, pages 127–138, 2019.

[12] Sebastian Kruse and Felix Naumann. Efficient discovery of approximate dependencies. *Proceedings of the VLDB Endowment*, 11(7):759–772, 2018.

[13] Thorsten Papenbrock and Felix Naumann. A hybrid approach to functional dependency discovery. In *Proceedings of the 2016 International Conference on Management of Data*, pages 821–833, 2016.

[14] Falco Dürsch, Axel Stebner, Fabian Windheuser, Maxi Fischer, Tim Friedrich, Nils Strelow, Tobias Bleifuß, Hazar Harmouch, Lan Jiang, Thorsten Papenbrock, et al. Inclusion dependency discovery: An experimental evaluation of thirteen algorithms. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, pages 219–228, 2019.

[15] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864, 2016.

[16] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. Learned cardinalities: Estimating correlated joins with deep learning. *arXiv preprint arXiv:1809.00677*, 2018.

[17] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. Deepdb: learn from data, not from queries! *arXiv preprint arXiv:1909.00607*, 2019.

[18] 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. *arXiv preprint arXiv:1905.04278*, 2019.

[19] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. Neurocard: one cardinality estimator for all tables. *arXiv preprint arXiv:2006.08109*, 2020.

[20] Rong Zhu, Ziniu Wu, Yuxing Han, Kai Zeng, Andreas Pfadler, Zhengping Qian, Jingren Zhou, and Bin Cui. Flat: Fast, lightweight and accurate method for cardinality estimation. *arXiv preprint arXiv:2011.09022*, 2020.

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

[22] Nikoli Dryden, Roman Böhringer, Tal Ben-Nun, and Torsten Hoefler. Clairvoyant prefetching for distributed machine learning i/o. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15, 2021.

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

[24] Melanie Weis, Felix Naumann, and Franziska Brosy. A duplicate detection benchmark for xml (and relational) data. In *Proc. of Workshop on Information Quality for Information Systems (IQIS)*, 2006.

[25] Korry Douglas and Susan Douglas. *PostgreSQL: a comprehensive guide to building, programming, and administering PostgresSQL databases*. SAMS publishing, 2003.

[26] P Krishnan, Jeffrey Scott Vitter, and Bala Iyer. Estimating alphanumeric selectivity in the presence of wildcards. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 282–293, 1996.

[27] HV Jagadish, Olga Kapitskaia, Raymond T Ng, and Divesh Srivastava. One-dimensional and multi-dimensional substring selectivity estimation. *The VLDB Journal*, 9:214–230, 2000.

[28] Franz Pernkopf. Bayesian network classifiers versus selective k-nn classifier. *Pattern recognition*, 38(1):1–10, 2005.

[29] Surajit Chaudhuri, Venkatesh Ganti, and Luis Gravano. Selectivity estimation for string predicates: Overcoming the underestimation problem. In *Proceedings. 20th International Conference on Data Engineering*, pages 227–238. IEEE, 2004.

[30] Suraj Shetiya, Saravanan Thirumuruganathan, Nick Koudas, and Gautam Das. Astrid: accurate selectivity estimation for string predicates using deep learning. *Proceedings of the VLDB Endowment*, 14(4), 2020.

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

[32] Jie Liu, Bogdan Nicolae, and Dong Li. Lobster: Load balance-aware i/o for distributed dnn training. In *Proceedings of the 51st International Conference on Parallel Processing*, pages 1–11, 2022.

[33] Jie Liu, Jiawen Liu, Zhen Xie, and Dong Li. Flame: A self-adaptive auto-labeling system for heterogeneous mobile processors. *arXiv preprint arXiv:2003.01762*, 2020.

[34] Jie Liu, Jiawen Liu, Wan Du, and Dong Li. Performance analysis and characterization of training deep learning models on mobile device. In *2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 506–515. IEEE, 2019.

[35] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. Analyzing and mitigating data stalls in DNN training. *arXiv preprint arXiv:2007.06775*, 2020.

[36] Abhishek Vijaya Kumar and Muthian Sivathanu. Quiver: An informed storage cache for deep learning. In *18th USENIX Conference on File and Storage Technologies*, pages 283–296, 2020.

[37] Mark Zhao, Niket Agarwal, Aarti Basant, Bugra Gedik, Satadru Pan, Mustafa Ozdal, Rakesh Komuravelli, Jerry Pan, Tianshu Bao, Haowei Lu, et al. Understanding and co-designing the data ingestion pipeline for industry-scale recsys training. *arXiv preprint arXiv:2108.09373*, 2021.

[38] Muhammad Adnan, Yassaman Ebrahimzadeh Maboud, Divya Mahajan, and Prashant J Nair. Accelerating recommendation system training by leveraging popular choices. *Proceedings of the VLDB Endowment*, 15(1):127–140, 2021.

[39] Ricardo Macedo, Cláudia Correia, Marco Dantas, Cláudia Brito, Weijia Xu, Yusuke Tanimura, Jason Haga, and Joao Paulo. The case for storage optimization decoupling in deep learning frameworks. In *IEEE International Conference on Cluster Computing*, pages 649–656. IEEE, 2021.

[40] Gyewon Lee, Irene Lee, Hyeonmin Ha, Kyunggeun Lee, Hwarim Hyun, Ahnjae Shin, and Byung-Gon Chun. Refurbish your training data: Reusing partially augmented samples for faster deep neural network training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 537–550, 2021.

[41] Jingru Yang, Ju Fan, Zhewei Wei, Guoliang Li, Tongyu Liu, and Xiaoyong Du. Cost-effective data annotation using game-based crowdsourcing. *Proceedings of the VLDB Endowment*, 12(1):57–70, 2018.

[42] Ahsanul Haque, Hemeng Tao, Swarup Chandra, Jie Liu, and Latifur Khan. A framework for multistream regression with direct density ratio estimation. In *Thirty-second AAAI conference on artificial intelligence*, 2018.

[43] Swarup Chandra, Ahsanul Haque, Hemeng Tao, Jie Liu, Latifur Khan, and Charu Aggarwal. Ensemble direct density ratio estimation for multistream classification. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 1364–1367. IEEE, 2018.

[44] Jie Liu, Wenqian Dong, Qingqing Zhou, and Dong Li. Fauce: Fast and accurate deep ensembles with uncertainty for cardinality estimation.

[45] Wenqian Dong, Jie Liu, Zhen Xie, and Dong Li. Adaptive neural network-based approximation to accelerate eulerian fluid simulation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–22, 2019.

[46] Ahsanul Haque, Latifur Khan, Michael Baron, Bhavani Thuraisingham, and Charu Aggarwal. Efficient handling of concept drift and concept evolution over stream data. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 481–492. IEEE, 2016.

[47] Zhuoyi Wang, Zelun Kong, Swarup Changra, Hemeng Tao, and Latifur Khan. Robust high dimensional stream classification with novel class detection. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1418–1429. IEEE, 2019.

[48] Petko Georgiev, Nicholas D Lane, Kiran K Rachuri, and Cecilia Mascolo. Dsp. ear: Leveraging co-processor support for continuous audio sensing on smartphones. In *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems*, pages 295–309, 2014.

[49] Daniel Maier, Nadjib Mammeri, Biagio Cosenza, and Ben Juurlink. Approximating memory-bound applications on mobile gpus. In *2019 International Conference on High Performance Computing & Simulation (HPCS)*, pages 329–335. IEEE, 2019.

[50] Qp solver. Quadratic programming solving kit. `https://github.com/hjkuijf/ALGLIB`.

[51] Snapdragon. qualcomm cpu sleep benchmarking. `https://developer.qualcomm.com/docs/snpe/benchmarking.html`.

[52] Yann Lecun. The mnist batabase. `http://yann.lecun.com/exdb/mnist/`.

[53] Gregory Cohen, Saeed Afshar, Jonathan Tapson, and Andre Van Schaik. Emnist: Extending mnist to handwritten letters. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 2921–2926. IEEE, 2017.

[54] toronto. The cifar100 batabase. `https://www.cs.toronto.edu/~kriz/cifar.html`.

[55] Oriol Vinyals, Charles Blundell, Timothy Lillicrap, Koray Kavukcuoglu, and Daan Wierstra. Matching networks for one shot learning. *arXiv preprint arXiv:1606.04080*, 2016.

[56] Kishore K Reddy and Mubarak Shah. Recognizing 50 human action categories of web videos. *Machine vision and applications*, 24(5):971–981, 2013.

[57] Khurram Soomro, Amir Roshan Zamir, and Mubarak Shah. Ucf101: A dataset of 101 human actions classes from videos in the wild. *arXiv preprint arXiv:1212.0402*, 2012.

[58] github. Ffmpeg software development kit. `https://github.com/tanersener/mobile-ffmpeg`.

[59] Bo Dong, Md Shihabul Islam, Swarup Chandra, Latifur Khan, and Bhavani Thuraisingham. Gci: A transfer learning approach for detecting cheats of computer game. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 1188–1197. IEEE, 2018.

[60] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[61] Sahin. Introduction to apple ml tools. In *Develop Intelligent iOS Apps with Swift*, pages 17–39. Springer, 2021.

[62] PassMark.2015. Passmark software - performancetest system benchmarks. `http://www.passmark.com/baselines/index.php`.

[63] Annamalai Narayanan, Mahinthan Chandramohan, Lihui Chen, Yang Liu, and Santhoshkumar Saminathan. subgraph2vec: Learning distributed representations of rooted sub-graphs from large graphs. *arXiv preprint arXiv:1606.08928*, 2016.

[64] David Lopez-Paz, Philipp Hennig, and Bernhard Schölkopf. The randomized dependence coefficient. *Advances in neural information processing systems*, 26:1–9, 2013.

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

[66] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S Sathiya Keerthi. An empirical analysis of deep learning for cardinality estimation. *arXiv preprint arXiv:1905.06425*, 2019.

[67] Surajit Chaudhuri, Gautam Das, and Vivek Narasayya. Optimized stratified sampling for approximate query processing. *ACM Transactions on Database Systems (TODS)*, 32(2):9–es, 2007.

[68] Jost Tobias Springenberg, Aaron Klein, Stefan Falkner, and Frank Hutter. Bayesian optimization with robust bayesian neural networks. *Advances in neural information processing systems*, 29:4134–4142, 2016.

[69] Yijun Xiao and William Yang Wang. Quantifying uncertainties in natural language processing tasks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 7322–7329, 2019.

[70] binghamton. Variance proof. `https://www2.math.binghamton.edu/lib/exe/fetch.php/people/renfrew/447-4-17.pdf`.

[71] Yarin Gal and Zoubin Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *international conference on machine learning*, pages 1050–1059, 2016.

[72] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[73] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[74] Byung-Jae Kwak, Nah-Oak Song, and Leonard E Miller. Performance analysis of exponential backoff. *IEEE/ACM transactions on networking*, 13(2):343–355, 2005.

[75] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. Stholes: a multidimensional workload-aware histogram. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 211–222, 2001.

[76] microsoft. queries contain correlations. `https://support.microsoft.com/en-us/topic/kb2658214-fix-poor-performance-when-you-run-a-query-that-contains`

[77] Walter Cai, Magdalena Balazinska, and Dan Suciu. Pessimistic cardinality estimation: Tighter upper bounds for intermediate join cardinalities. In *Proceedings of the 2019 International Conference on Management of Data*, pages 18–35, 2019.

[78] Rojeh Hayek and Oded Shmueli. Nn-based transformation of any sql cardinality estimator for handling distinct, and, or and not. *arXiv preprint arXiv:2004.07009*, 2020.

[79] Yka Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. Tane: An efficient algorithm for discovering functional and approximate dependencies. *The computer journal*, 42(2):100–111, 1999.

[80] Ziawasch Abedjan, Jorge-Arnulfo Quiané-Ruiz, and Felix Naumann. Detecting unique column combinations on dynamic data. In *2014 IEEE 30th International Conference on Data Engineering*, pages 1036–1047. IEEE, 2014.

[81] Peter A Flach and Iztok Savnik. Database dependency discovery: a machine learning approach. *AI communications*, 12(3):139–160, 1999.

[82] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems*, 32:8026–8037, 2019.

[83] Mahdi Zolnouri, Xinlin Li, and Vahid Partovi Nia. Importance of data loading pipeline in training dnns. *arXiv preprint arXiv:2005.02130*, 2020.

[84] Dong Li, Bronis de Supinski, Martin Schulz, Dimitrios S. Nikolopoulos, and Kirk W. Cameron. Hybrid MPI/OpenMP power-aware computing. In *International Parallel and Distributed Processing Symposium*, 2010.

[85] Matthew Curtis-Maury, Ankur Shah, Filip Blagojevic, Dimitrios S. Nikolopoulos, Bronis R. de Supinski, and Martin Schulz. Prediction models for multi-dimensional power-performance optimization on many cores. In *International Conference on Parallel Architectures and Compilation Techniques*, 2008.

[86] Vasilis Sourlas, Lazaros Gkatzikis, Paris Flegkas, and Leandros Tassiulas. Distributed cache management in information-centric networks. *IEEE Transactions on Network and Service Management*, 10(3):286–299, 2013.

[87] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[88] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. ShuffleNet: An extremely efficient convolutional neural network for mobile devices. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 6848–6856, 2018.

[89] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[90] A Handa, V Patraucean, V Badrinarayanan, S Stent, and R Cipolla. SceneNet: Understanding real world indoor scenes with synthetic data. *arXiv preprint arXiv:1511.07041*, 2015.

[91] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[92] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

[93] Tal Ridnik, Emanuel Ben-Baruch, Asaf Noy, and Lihi Zelnik-Manor. Imagenet-21k pretraining for the masses. *arXiv preprint arXiv:2104.10972*, 2021.

[94] Luca Bonomi, Li Xiong, Rui Chen, and Benjamin CM Fung. Frequent grams based embedding for privacy preserving record linkage. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 1597–1601, 2012.

[95] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. Preventing bad plans by bounding the impact of cardinality estimation errors. *Proceedings of the VLDB Endowment*, 2(1):982–993, 2009.

[96] Tugkan Batu, Funda Ergun, and Cenk Sahinalp. Oblivious string embeddings and edit distance approximations. In *SODA*, volume 6, pages 792–801, 2006.

[97] Francesco Pappalardo, Cristiano Calonaci, Marzio Pennisi, Emilio Mastriani, and Santo Motta. Hamfast: fast hamming distance computation. In *2009 WRI World Congress on Computer Science and Information Engineering*, volume 1, pages 569–572. IEEE, 2009.

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

[99] Fionn Murtagh and Pierre Legendre. Ward's hierarchical clustering method: clustering criterion and agglomerative algorithm. *arXiv preprint arXiv:1111.6285*, 2011.

[100] Yonglong Tian, Chen Sun, Ben Poole, Dilip Krishnan, Cordelia Schmid, and Phillip Isola. What makes for good views for contrastive learning? *Advances in neural information processing systems*, 33:6827–6839, 2020.

[101] Ziyu Jiang, Tianlong Chen, Bobak J Mortazavi, and Zhangyang Wang. Self-damaging contrastive learning. In *International Conference on Machine Learning*, pages 4927–4939. PMLR, 2021.

[102] Kihyuk Sohn. Improved deep metric learning with multi-class n-pair loss objective. *Advances in neural information processing systems*, 29, 2016.

[103] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 2623–2631, 2019.

[104] Tian Tian and Jun Zhu. Max-margin majority voting for learning from crowds. *Advances in neural information processing systems*, 28, 2015.

[105] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[106] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, 2017.

[107] Hongrae Lee, Raymond T Ng, and Kyuseok Shim. Approximate substring selectivity estimation. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, pages 827–838, 2009.

[108] Arturas Mazeika, Michael H Böhlen, Nick Koudas, and Divesh Srivastava. Estimating the selectivity of approximate string queries. *ACM Transactions on Database Systems (TODS)*, 32(2):12–es, 2007.

[109] Noga Alon, Martin Dietzfelbinger, Peter Bro Miltersen, Erez Petrank, and Gábor Tardos. Linear hash functions. *Journal of the ACM (JACM)*, 46(5):667–683, 1999.

[110] Edith Cohen. Size-estimation framework with applications to transitive closure and reachability. *Journal of Computer and System Sciences*, 55(3):441–453, 1997.

[111] Andreas Kipf, Dimitri Vorona, Jonas Müller, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, Thomas Neumann, and Alfons Kemper. Estimating cardinalities with deep sketches. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1937–1940, 2019.

[112] Yaoshu Wang, Chuan Xiao, Jianbin Qin, Xin Cao, Yifang Sun, Wei Wang, and Makoto Onizuka. Monotonic cardinality estimation of similarity selection: A deep learning approach. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1197–1212, 2020.

[113] Suyong Kwon, Woohwan Jung, and Kyuseok Shim. Cardinality estimation of approximate substring queries using deep learning. *Proceedings of the VLDB Endowment*, 15(11):3145–3157, 2022.

[114] Long Short-Term Memory. Long short-term memory. *Neural computation*, 9(8):1735–1780, 2010.

[115] Mehmet Aytimur and Ali Cakmak. Using positional sequence patterns to estimate the selectivity of sql like queries. *Expert Systems with Applications*, 165:113762, 2021.

[116] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. Bao: Making learned query optimization practical. In *Proceedings of the 2021 International Conference on Management of Data*, pages 1275–1288, 2021.

[117] Yue Zhao, Gao Cong, Jiachen Shi, and Chunyan Miao. Queryformer: a tree transformer model for query plan representation. *Proceedings of the VLDB Endowment*, 15(8):1658–1670, 2022.

[118] Zongheng Yang, Wei-Lin Chiang, Sifei Luan, Gautam Mittal, Michael Luo, and Ion Stoica. Balsa: Learning a query optimizer without expert demonstrations. In *Proceedings of the 2022 International Conference on Management of Data*, pages 931–944, 2022.

[119] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. *Advances in neural information processing systems*, 24, 2011.

[120] Jure Leskovec, Lars Backstrom, and Jon Kleinberg. Meme-tracking and the dynamics of the news cycle. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 497–506, 2009.

[121] Youyun Wang, Chuzhe Tang, Zhaoguo Wang, and Haibo Chen. Sindex: a scalable learned index for string keys. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 17–24, 2020.

[122] Benjamin Spector, Andreas Kipf, Kapil Vaidya, Chi Wang, Umar Farooq Minhas, and Tim Kraska. Bounding the last mile: Efficient learned string indexing. *arXiv preprint arXiv:2111.14905*, 2021.

[123] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 183–196, 2012.

[124] Xingbo Wu, Fan Ni, and Song Jiang. Wormhole: A fast ordered index for in-memory data management. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16, 2019.

[125] Mean Absolute Error. Mean absolute error. *Retrieved September*, 19:2016, 2016.

[126] Jiangpeng He, Runyu Mao, Zeman Shao, and Fengqing Zhu. Incremental learning in online scenario. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 13926–13935, 2020.

[127] Bo Dong, Cristian Lumezanu, Yuncong Chen, Dongjin Song, Takehiko Mizoguchi, Haifeng Chen, and Latifur Khan. At the speed of sound: Efficient audio scene classification. In *Proceedings of the 2020 International Conference on Multimedia Retrieval*, pages 301–305, 2020.

[128] Fulton Wang and Cynthia Rudin. Falling rule lists. In *Artificial Intelligence and Statistics*, pages 1013–1022, 2015.

[129] Tong Wang, Cynthia Rudin, Finale Doshi-Velez, Yimin Liu, Erica Klampfl, and Perry MacNeille. Or's of and's for interpretable classification, with application to context-aware recommender systems. *arXiv preprint arXiv:1504.07614*, 2015.

[130] Xueqing Deng, Yi Zhu, Yuxin Tian, and Shawn Newsam. Scale aware adaptation for land-cover classification in remote sensing imagery. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, pages 2160–2169, 2021.

[131] Bo Dong, Yifan Li, Yang Gao, Ahsanul Haque, Latifur Khan, and Mohammad M Masud. Multistream regression with asynchronous concept drift detection. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 596–605. IEEE, 2017.

[132] Yuxin Tian, Xueqing Deng, Yi Zhu, and Shawn Newsam. Cross-time and orientation-invariant overhead image geolocalization using deep local features. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, pages 2512–2520, 2020.

[133] Stephen H Bach, Bryan He, Alexander Ratner, and Christopher Ré. Learning the structure of generative models without labeled data. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 273–282. JMLR. org, 2017.

[134] Paroma Varma, Bryan He, Payal Bajaj, Imon Banerjee, Nishith Khandwala, Daniel L Rubin, and Christopher Ré. Inferring generative model structure with static analysis. *Advances in neural information processing systems*, 30:239, 2017.

[135] Bo Dong, Yang Gao, Swarup Chandra, and Latifur Khan. Multistream classification with relative density ratio estimation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 3478–3485, 2019.

[136] Trevor Hastie, Saharon Rosset, Ji Zhu, and Hui Zou. Multi-class adaboost. *Statistics and its Interface*, 2(3):349–360, 2009.

[137] Karl Weiss, Taghi M Khoshgoftaar, and DingDing Wang. A survey of transfer learning. *Journal of Big data*, 3(1):9, 2016.

[138] William Yang Wang, Kathryn Mazaitis, and William W Cohen. Structure learning via parameter learning. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, pages 1199–1208, 2014.

[139] Manas Joglekar, Hector Garcia-Molina, and Aditya Parameswaran. Comprehensive and reliable crowd assessment algorithms. In *2015 IEEE 31st International Conference on Data Engineering*, pages 195–206. IEEE, 2015.

[140] Victor S Sheng, Foster Provost, and Panagiotis G Ipeirotis. Get another label? improving data quality and data mining using multiple, noisy labelers. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 614–622. ACM, 2008.

[141] Ni Lao and William W Cohen. Relational retrieval using a combination of path-constrained random walks. *Machine learning*, 81(1):53–67, 2010.

[142] Zhen Xie, Wenqian Dong, Jiawen Liu, Hang Liu, and Dong Li. Tahoe: tree structure-aware high performance inference engine for decision tree ensemble on gpu. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 426–440, 2021.

[143] Jiawen Liu, Jie Ren, Roberto Gioiosa, Dong Li, and Jiajia Li. Sparta: High-performance, element-wise sparse tensor contraction on heterogeneous memory. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 318–333, 2021.

[144] Bang Di, Jiawen Liu, Hao Chen, and Dong Li. Fast, flexible, and comprehensive bug detection for persistent memory programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 503–516, 2021.

[145] Jiawen Liu, Dong Li, Roberto Gioiosa, and Jiajia Li. Athena: high-performance sparse tensor contraction sequence on heterogeneous memory. In *Proceedings of the ACM International Conference on Supercomputing*, pages 190–202, 2021.

[146] Xin He, Jiawen Liu, Zhen Xie, Hao Chen, Guoyang Chen, Weifeng Zhang, and Dong Li. Enabling energy-efficient dnn training on hybrid gpu-fpga accelerators. In *Proceedings of the ACM International Conference on Supercomputing*, pages 227–241, 2021.

[147] Jiawen Liu, Zhen Xie, Dimitrios Nikolopoulos, and Dong Li. {RIANN}: Real-time incremental learning with approximate nearest neighbor on mobile devices. In *2020 {USENIX} Conference on Operational Machine Learning (OpML 20)*, 2020.

[148] Robert LiKamWa and Lin Zhong. Starfish: Efficient concurrency support for computer vision applications. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 213–226. ACM, 2015.

[149] Nicholas D Lane, Sourav Bhattacharya, Petko Georgiev, Claudio Forlivesi, Lei Jiao, Lorena Qendro, and Fahim Kawsar. Deepx: A software accelerator for low-power deep learning inference on mobile devices. In *Proceedings of the 15th International Conference on Information Processing in Sensor Networks (IPSN)*, page 23. IEEE Press, 2016.

[150] Jiawen Liu, Dong Li, Gokcen Kestor, and Jeffrey Vetter. Runtime concurrency control and operation scheduling for high performance neural network training. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 188–199. IEEE, 2019.

[151] Samuel S Ogden and Tian Guo. {MODI}: Mobile deep inference made efficient by edge computing. In *Workshop on Hot Topics in Edge Computing (HotEdge)*, 2018.

[152] Songtao Guo, Bin Xiao, Yuanyuan Yang, and Yang Yang. Energy-efficient dynamic offloading and resource scheduling in mobile cloud computing. In *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9. IEEE, 2016.

[153] Ariel Gordon, Elad Eban, Ofir Nachum, Bo Chen, Hao Wu, Tien-Ju Yang, and Edward Choi. Morphnet: Fast & simple resource-constrained structure learning of deep networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1586–1595, 2018.

[154] Dawei Li, Xiaolong Wang, and Deguang Kong. Deeprebirth: Accelerating deep neural network execution on mobile devices. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[155] Wei Niu, Xiaolong Ma, Sheng Lin, Shihao Wang, Xuehai Qian, Xue Lin, Yanzhi Wang, and Bin Ren. Patdnn: Achieving real-time dnn execution on mobile devices with pattern-based weight pruning. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 907–922, 2020.

[156] Hongjia Li, Ning Liu, Xiaolong Ma, Sheng Lin, Shaokai Ye, Tianyun Zhang, Xue Lin, Wenyao Xu, and Yanzhi Wang. Admm-based weight pruning for real-time deep learning acceleration on mobile devices. In *Proceedings of the 2019 on Great Lakes Symposium on VLSI*, pages 501–506, 2019.

[157] Xiaolong Ma, Fu-Ming Guo, Wei Niu, Xue Lin, Jian Tang, Kaisheng Ma, Bin Ren, and Yanzhi Wang. Pconv: The missing but desirable sparsity in dnn weight pruning for real-time execution on mobile devices. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 5117–5124, 2020.

[158] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 615–629. ACM, 2017.

[159] En Li, Zhi Zhou, and Xu Chen. Edge intelligence: On-demand deep learning model co-inference with device-edge synergy. In *Proceedings of the 2018 Workshop on Mobile Edge Communications*, pages 31–36, 2018.

[160] Hyuk-Jin Jeong, Hyeon-Jae Lee, Chang Hyun Shin, and Soo-Mook Moon. Ionn: Incremental offloading of neural network computations from mobile devices to edge servers. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 401–411, 2018.

[161] Biyi Fang, Xiao Zeng, and Mi Zhang. Nestdnn: Resource-aware multi-tenant on-device deep learning for continuous mobile vision. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking (MobiCom)*, pages 115–127. ACM, 2018.

[162] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. Amc: Automl for model compression and acceleration on mobile devices. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 784–800, 2018.

[163] Sicong Liu, Yingyan Lin, Zimu Zhou, Kaiming Nan, Hui Liu, and Junzhao Du. On-demand deep model compression for mobile devices: A usage-driven model selection framework. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 389–400. ACM, 2018.

[164] Petko Georgiev, Nicholas D Lane, Kiran K Rachuri, and Cecilia Mascolo. Leo: Scheduling sensor inference algorithms across heterogeneous mobile processors and network resources. In *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking (MobiCom)*, pages 320–333. ACM, 2016.

[165] Jiannan Wang, Tim Kraska, Michael J Franklin, and Jianhua Feng. Crowder: Crowdsourcing entity resolution. *Proceedings of the VLDB Endowment*, 5(11):1483–1494, 2012.

[166] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[167] Yongjoo Park, Ahmad Shahab Tajik, Michael Cafarella, and Barzan Mozafari. Database learning: Toward a database that becomes smarter every time. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 587–602, 2017.

[168] Utkarsh Srivastava, Peter J Haas, Volker Markl, Marcel Kutsch, and Tam Minh Tran. Isomer: Consistent histogram construction using query feedback. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 39–39. IEEE, 2006.

[169] Michael Stillger, Guy M Lohman, Volker Markl, and Mokhtar Kandil. Leo-db2's learning optimizer. In *VLDB*, volume 1, pages 19–28, 2001.

[170] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1009–1024, 2017.

[171] Max Heimel, Martin Kiefer, and Volker Markl. Self-tuning, gpu-accelerated kernel density models for multidimensional selectivity estimation. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1477–1492, 2015.

[172] Martin Kiefer, Max Heimel, Sebastian Breß, and Volker Markl. Estimating join selectivities using bandwidth-optimized kernel density models. *Proceedings of the VLDB Endowment*, 10(13):2085–2096, 2017.

[173] James Martens and Venkatesh Medabalimi. On the expressive efficiency of sum product networks. *arXiv preprint arXiv:1411.7717*, 2014.

[174] Hoifung Poon and Pedro Domingos. Sum-product networks: A new deep architecture. In *2011 IEEE International Conference on Computer Vision Workshops (ICCV Workshops)*, pages 689–690. IEEE, 2011.

[175] Kostas Tzoumas, Amol Deshpande, and Christian S Jensen. Lightweight graphical models for selectivity estimation without independence assumptions. *Proceedings of the VLDB Endowment*, 4(11):852–863, 2011.

[176] Conor Durkan and Charlie Nash. Autoregressive energy machines. In *ICML*, 2019.

[177] Mathieu Germain, Karol Gregor, Iain Murray, and Hugo Larochelle. Made: Masked autoencoder for distribution estimation. In *International Conference on Machine Learning*, pages 881–889, 2015.

[178] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. Learning multi-dimensional indexes. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 985–1000, 2020.

[179] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, pages 489–504, 2018.

[180] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 270–288. 2019.

[181] Idan Burstein. Nvidia data center processing unit (dpu) architecture. In *2021 IEEE Hot Chips 33 Symposium (HCS)*, pages 1–20. IEEE, 2021.

[182] Pyeongsu Park, Heetaek Jeong, and Jangwoo Kim. Trainbox: An extreme-scale neural network training server architecture by systematically balancing operations. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 825–838. IEEE, 2020.

[183] Daniel Povey, Xiaohui Zhang, and Sanjeev Khudanpur. Parallel training of deep neural networks with natural gradient and parameter averaging. *arXiv preprint arXiv:1410.7455*, 2014.

[184] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. Pytorch distributed: Experiences on accelerating data parallel training. *arXiv preprint arXiv:2006.15704*, 2020.

[185] Soojeong Kim, Gyeong-In Yu, Hojin Park, Sungwoo Cho, Eunji Jeong, Hyeonmin Ha, Sanha Lee, Joo Seong Jeong, and Byung-Gon Chun. Parallax: Sparsity-aware data parallel training of deep neural networks. In *15th EuroSys Conference*, pages 1–15, 2019.

[186] Chien-Chin Huang, Gu Jin, and Jinyang Li. Swapadvisor: Pushing deep learning beyond the gpu memory limit via smart swapping. In *25th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1341–1355, 2020.

[187] Nikoli Dryden, Tim Moon, Sam Ade Jacobs, and Brian Van Essen. Communication quantization for data-parallel training of deep neural networks. In *2016 2nd Workshop on Machine Learning in HPC Environments (MLHPC)*, pages 1–8. IEEE, 2016.

[188] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. Zero-offload: Democratizing billion-scale model training. *arXiv preprint arXiv:2101.06840*, 2021.

[189] Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. *arXiv preprint arXiv:2104.07857*, 2021.

[190] Zhen Xie, Jie Liu, Jiajia Li, and Dong Li. Merchandiser: Data placement on heterogeneous memory for task-parallel hpc applications with load-balance awareness. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pages 204–217, 2023.

[191] Jie Liu, Bogdan Nicolae, Dong Li, Justin M Wozniak, Tekin Bicer, Zhengchun Liu, and Ian Foster. Large scale caching and streaming of training data for online deep learning. In *Proceedings of the 12th Workshop on AI and Scientific Computing at Scale using Flexible Computing Infrastructures*, pages 19–26, 2022.

[192] Wei Shu and Nian-Feng Tzeng. Relinquishment coherence for enhancing directory efficiency in chip multiprocessors. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*, pages 372–375. IEEE, 2016.

[193] Deepak Narayanan, Keshav Santhanam, and Matei Zaharia. Accelerating model search with model batching. In *1st Conference on Systems and Machine Learning (SysML), SysML*, volume 18, 2018.

[194] Aarati Kakaraparthy, Abhay Venkatesh, Amar Phanishayee, and Shivaram Venkataraman. The case for unifying data loading in machine learning clusters. In *11th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, 2019.

[195] Wei Shu and Nian-Feng Tzeng. Compressed sharer tracking and relinquishment coherence for superior directory efficiency of chip multiprocessors. *IEEE Transactions on Computers*, 66(11):1975–1981, 2017.

[196] Wei Shu and Nian-Feng Tzeng. Nuda: Non-uniform directory architecture for scalable chip multiprocessors. *IEEE Transactions on Computers*, 67(5):740–747, 2017.

[197] Suyash Mahar, Hao Wang, Wei Shu, and Abhishek Dhanotia. Workload behavior driven memory subsystem design for hyperscale. *arXiv preprint arXiv:2303.08396*, 2023.

[198] Supun Nakandala, Yuhao Zhang, and Arun Kumar. Cerebro: A data system for optimized deep learning model selection. *Proceedings of the VLDB Endowment*, 13(12):2159–2173, 2020.

[199] Yue Zhu, Fahim Chowdhury, Huansong Fu, Adam Moody, Kathryn Mohror, Kento Sato, and Weikuan Yu. Entropy-aware i/o pipelining for large-scale deep learning on hpc systems. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 145–156. IEEE, 2018.

[200] Lipeng Wang, Songgao Ye, Baichen Yang, Youyou Lu, Hequan Zhang, Shengen Yan, and Qiong Luo. Diesel: A dataset-based distributed storage and caching system for large-scale deep learning training. In *49th International Conference on Parallel Processing*, pages 1–11, 2020.

[201] Zhiyuan Chen, Nick Koudas, Flip Korn, and Shanmugavelayutham Muthukrishnan. Selectively estimation for boolean queries. In *Proceedings of the nineteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 216–225, 2000.

[202] Mehmet Aytimur and Ali Cakmak. Estimating the selectivity of like queries using pattern-based histograms. *Turkish Journal of Electrical Engineering and Computer Sciences*, 26(6):3319–3334, 2018.

[203] Xiaochun Yang, Bin Wang, and Chen Li. Cost-based variable-length-gram selection for string collections to support approximate queries efficiently. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 353–364, 2008.

[204] David Kosiur. *Understanding Policy-Based Networking*. Wiley, New York, NY, 2nd. edition, 2001.

[205] Donald E. Knuth. *The Art of Computer Programming*, volume 1 of *Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., 3rd edition, 1998. (book).

[206] Xiang Yu, Chengliang Chai, Guoliang Li, and Jiabin Liu. Cost-based or learning-based? a hybrid query optimizer for query plan selection. *Proceedings of the VLDB Endowment*, 15(13):3924–3936, 2022.

[207] Rong Zhu, Ziniu Wu, Chengliang Chai, Andreas Pfadler, Bolin Ding, Guoliang Li, and Jingren Zhou. Learned query optimizer: At the forefront of ai-driven databases. In *EDBT*, pages 1–4, 2022.

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

[209] Kangfei Zhao, Jeffrey Xu Yu, Zongyan He, Rui Li, and Hao Zhang. Lightweight and accurate cardinality estimation by neural network gaussian process. In *Proceedings of the 2022 International Conference on Management of Data*, pages 973–987, 2022.

[210] Jin Chen, Guanyu Ye, Yan Zhao, Shuncheng Liu, Liwei Deng, Xu Chen, Rui Zhou, and Kai Zheng. Efficient join order selection learning with graph-based representation. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 97–107, 2022.

[211] Parimarjan Negi, Ryan Marcus, Andreas Kipf, Hongzi Mao, Nesime Tatbul, Tim Kraska, and Mohammad Alizadeh. Flow-loss: Learning cardinality estimates that matter. *arXiv preprint arXiv:2101.04964*, 2021.

[212] Xiaoying Wang, Changbo Qu, Weiyuan Wu, Jiannan Wang, and Qingqing Zhou. Are we ready for learned cardinality estimation? *arXiv preprint arXiv:2012.06743*, 2020.