Performance-Based Service Level Agreements for Data Analytics in the Cloud

Jennifer Ortiz

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2019

Reading Committee:

Magdalena Balazinska, Chair

Dan Suciu

Hannaneh Hajishirzi

Program Authorized to Offer Degree:
Computer Science and Engineering

University of Washington

Abstract

Performance-Based Service Level Agreements
for Data Analytics in the Cloud

Jennifer Ortiz

Chair of the Supervisory Committee:
Professor Magdalena Balazinska
Computer Science and Engineering

A variety of data analytics systems are available as cloud services today, such as Amazon Elastic MapReduce (EMR) [13] and Azure Data Lake Analytics [1]. To buy these services, users select and pay for a given cluster configuration based on the number and type of service instances.

Today's cloud service pricing models force users to translate their data management needs into resource needs. It is well known, however, that users have difficulty selecting a configuration that meets their need. For non-experts, being faced with decisions about the configuration is even harder, especially when they seek to explore a new dataset.

This thesis focuses on the challenges and implementation details of building a system that helps bridge the gap between the data analytics services users need and the way cloud providers offer them.

The first challenge in closing the gap is finding a new type of abstraction that simplifies user interactions with cloud services. We introduce the notion of a "Personalized Service Level Agreement" (PSLA) and the PSLAManager system that implements it. Instead of asking users to specify the exact resources they think they need or asking them for exact queries that must be executed, PSLAManager shows them service options for a set price.

Second, providing PSLAs is challenging to service providers who seek to avoid paying for SLA violations and over-provisioning their resources. To address these challenges, we present

SLAOrchestrator, a system that supports performance-centric (rather than availability-centric) SLAs for data analytic services. SLAOrchestrator uses PSLAManager to generate SLAs; however, to reduce their resource and SLA violation costs, it introduces a new sub-system called PerfEnforce that uses scaling algorithms and provisioning techniques to minimize the financial penalties to providers.

Finally, SLAOrchestrator's models rely, among others, on the cost estimates from the query optimizer of the data analytics service. With more complex workloads, these estimates can be inaccurate due to the cardinality estimation problem. In the DeepQuery project, we empirically evaluate how deep learning can improve the accuracy of cardinality estimation.

TABLE OF CONTENTS

Page

LIST OF FIGURES

Figure Number Page

LIST OF TABLES

Table Number Page

ACKNOWLEDGMENTS

I feel incredibly blessed to have so many friends and loved ones to thank for the making of this thesis. This work would not be complete without them.

I would first like to thank my Ph.D. advisor, Magdalena Balazinska. Ever since my first research meeting with her, Magda has been very patient and encouraging every step of the way. Her enthusiasm for my project was one of the motivators that kept me going. She not only provided advice about database research, but also about more important lessons in life and what it means to be successful. I could not have asked for a better advisor.

I would also like to thank Dan Suciu for joining my committee and for his support on my research. I will always remember my first meeting with Dan during visit days, where he was quite eager to teach me about Datalog. His excitement for research and teaching is inspiring.

I thank all of my co-authors. Johannes Gehrke, for providing feedback and for helping me during my job search. Sathiya Keerthi, for patiently teaching me about machine learning concepts that were difficult for me to understand. Sarah Loebman, for providing a new perspective on the challenges of data management in astronomy. I would also like to thank Victor Almeida for mentoring me during my first couple years of graduate school. A special thanks to the great undergraduate students I had the chance to work with: Lee Lee Choo, Brenden Lee, and Khanh Duc Tran.

A huge thanks to the UWDB group who have made graduate school a fulfilling experience: Maaz Ahmad, Shumo Chu, Maureen Daum, Brandon Haynes, Srini Iyer, Shrainik Jain, Tomer Kaftan, Dominik Moritz, Ryan Maas, Parmita Mehta, Laurel Orr, Guna Prasaad, Shana Hutchison, Walter Cai, and Cong Yan. Also a special thanks to all those who have since graduated and have passed down their wisdom: Emad Soroush, Jingjing Wang, Kristi Morton, Dan Halperin, Paris Koutris, Nodira Khoussainova, and Prasang Upadhyaya.

I would like to thank the underground database reading group (DBURG). I thank this group for making the generals exam a less frightening experience. Seeing everyone struggle throughout the difficult readings certainly made me feel less alone in the process.

I could not have even started graduate school without help from many other amazing instructors and teachers. During my years at Ironwood, I thank Ms. Ciezki for teaching me how to program for the first time and for being there during my first conference talk. I also thank Ms. Pike, Ms. Pitzer, and Ms. White for giving me valuable career advice.

I learned about my passion for research thanks to the professors and students at Arizona State University. I thank Dr. Omayra Ortega for introducing me to research for the first time through the MERG group. I'm thankful that she and Dr. Yasin Silva encouraged me to apply for more Ph.D. programs, even though I did not have enough confidence to apply. I thank Dr. Suzanne Dietrich for introducing me to the world of database research and for helping me write my first research paper. I feel grateful about how she would always take the time to carefully guide me through each research project and inspired me to consider graduate school. I thank Mahesh Chaudhari for being the first to show me the joys and hardships of being a Ph.D. student.

Last, but certainly not least, I would like to thank my family. Many thanks to Melissa Ortiz and Wesley Luna for our movie and board game nights. Although at times it felt difficult to take a break from work, I always felt glad that I did. I am especially thankful that Melissa has always been patient, understanding, and able to listen when I needed it the most.

I thank Rafael Rivera for supporting me, pushing me, and believing in me. He has always been there to listen during frustrating moments and could easily make me laugh when I've wanted to cry. I thank him for occasionally reminding me that there is much more to life than just "work".

I do not think I can fully express the amount of gratitude and admiration I have for my parents. I thank both of my parents for believing in me and pushing me to be my best. Mom, thank you for your on-going support, kindness, and love. Dad, thank you for giving me advice about life, career, and even my research. You're a true engineer at heart.

Chapter 1

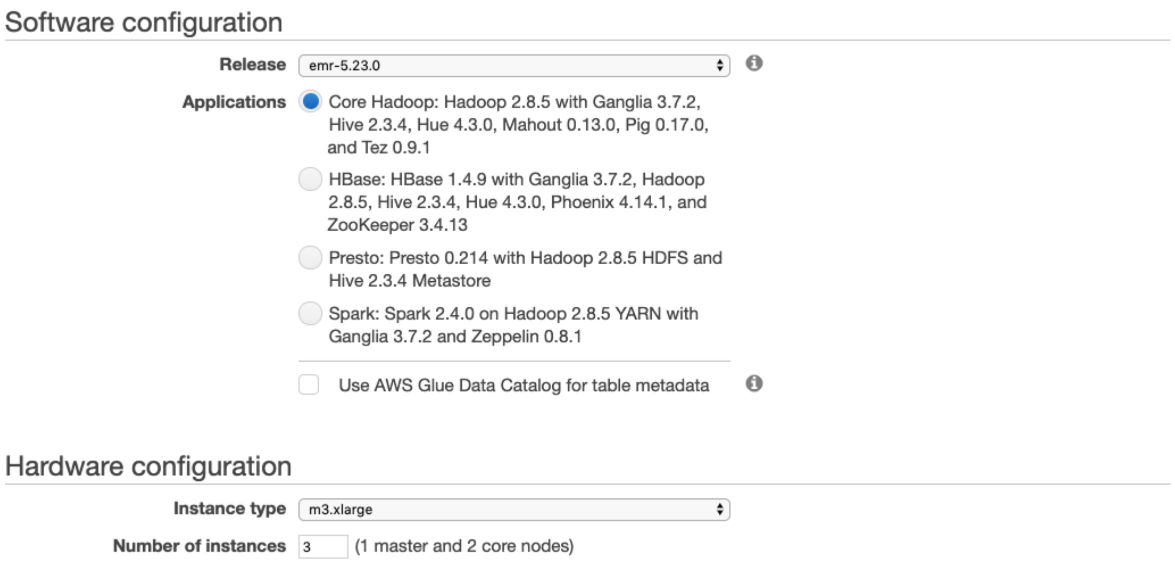
INTRODUCTION

Many of today's large enterprises collect massive amounts of data. Amazon, for example, stores more than 50 petabytes of data and processes over 600 thousand user analytic jobs daily on its analytics infrastructure [10]. Similarly, the Netflix data warehouse contains over 60 petabytes of data and processes more than 500 billion events per day [103]. As data continues to accumulate, large enterprises constantly seek new and efficient means to store and analyze it.

Data analytics refers to the science of analyzing data to uncover useful insights [90]. It commonly involves the handling and processing of large amounts of raw data, requiring computational power substantially beyond the reach of a typical desktop environment [67]. In fact, a vast pool of resources may be required and enterprises must choose between purchasing and maintaining them on-premise or using public cloud resources.

Public clouds are computing services offered by *cloud providers* over the Internet. Cloud providers make their compute infrastructure available to customers, who either rent basic virtual machines per hour or purchase a higher level data analytics service [133]. Compared to having an on-premise infrastructure, these options typically involve convenient, on-demand access to configurable computing resources [113]. Amazon Web Services [11], Microsoft Azure [134], and Google Cloud [54] are some of the largest cloud computing providers today.

These cloud providers offer *cloud storage* services including Amazon S3 [15], Amazon Elastic Block Store [9], Azure Blob Storage [19], and Google Cloud Storage [53]. They also offer general-purpose compute services, such as Amazon EC2 [12], Azure Virtual Machines [20] and Google Compute [54]. They also offer *data analytics* services including Amazon Elastic MapReduce (EMR) [13], Google BigQuery [21] and Azure Data Lake Analytics [1]. Users can simply choose this pre-packaged software to easily run queries and user-defined computations without having to



Software configuration

Release: emr-5.23.0

Applications:

● Core Hadoop: Hadoop 2.8.5 with Ganglia 3.7.2, Hive 2.3.4, Hue 4.3.0, Mahout 0.13.0, Pig 0.17.0, and Tez 0.9.1

○ HBase: HBase 1.4.9 with Ganglia 3.7.2, Hadoop 2.8.5, Hive 2.3.4, Hue 4.3.0, Phoenix 4.14.1, and ZooKeeper 3.4.13

○ Presto: Presto 0.214 with Hadoop 2.8.5 HDFS and Hive 2.3.4 Metastore

○ Spark: Spark 2.4.0 on Hadoop 2.8.5 YARN with Ganglia 3.7.2 and Zeppelin 0.8.1

☐ Use AWS Glue Data Catalog for table metadata

Hardware configuration

Instance type: m3.xlarge

Number of instances: 3 (1 master and 2 core nodes)

Figure 1.1: Amazon Elastic Map Reduce (EMR) Interface: Before using the service, users must select the software configuration and hardware configuration.

write their own code to run these operations.

Today's data analytic services require users to select the number of compute instances they want to purchase to satisfy their processing needs. We show a portion of this interface in Figure 1.1, where the user must select a software and hardware configuration. The number of choices is large. For example, as of April 2019, there were 115 compute instance types from Amazon Elastic MapReduce (EMR) [13] to select from. Each *instance type* is a machine that contains a specified amount of RAM, a select number of CPU cores and a storage type (either local, remote, or both). As shown in Figure 1.1, today's cloud services force users to translate their data management needs into resource needs (e.g., to answer such questions as: What type of instance should I use? How many instances should I purchase?).

A disconnect, thus exists between the resource-centric approach offered by cloud providers and the query-needs-based approach from users. Understanding the resources needed to process data management workloads can be daunting – particularly when users do not clearly understand their

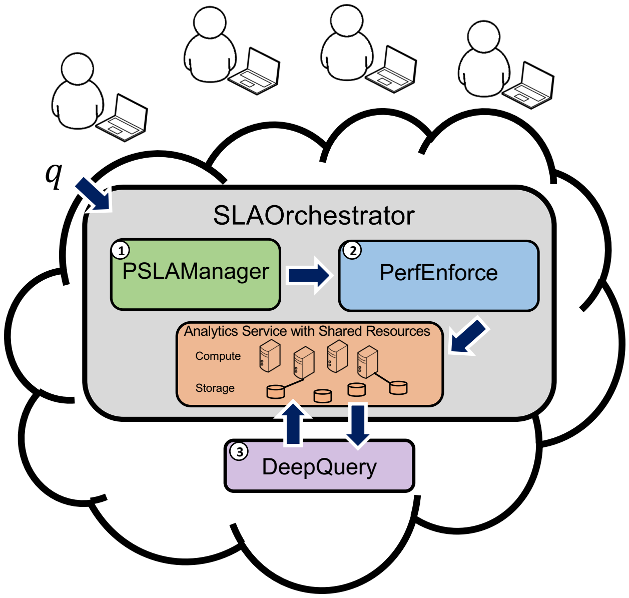


Figure 1.2: Overview of thesis contributions: A system that provides performance-based SLAs for data analytics in the cloud.

dataset or even know what results they seek, as is the case with exploratory analytics. Further, the service configuration of choice dramatically impacts price [8] and performance [129], compounding the pressure on users to choose correctly [60].

As a motivating example, assume a user decides to purchase an Amazon EMR cluster with 1 m3.xLarge master node and 2 m3.xLarge workers. At present, the cost would be $.99 per hour for the cluster. As the user begins to run queries, the question of performance arises: although queries return answers, some run faster than others. So, the user may begin to question whether purchasing more machines could improve overall performance [95].

At present, users do not know how to answer this question. Most cloud providers offer service level agreement (SLA) guarantees with respect to *availability*, but not with respect to *performance*. Without performance guarantees, users cannot make well-informed decisions about optimizing for cost and performance.

We posit that accessing a database management system (DBMS) as a cloud service opens the opportunity to re-think the contract between users and resource suppliers. This would be especially beneficial for data scientists with different skill levels (from data enthusiasts to statisticians) who need to manage and analyze their data but lack the expertise to determine the resource configuration they should use. How can they intelligently choose between even basic service instances, let alone decide about advanced services, such as the use of Spot instances [11]? This limits how many users can *cost-effectively* leverage a cloud DBMS service to manage and analyze their data.

This problem is difficult not only for data scientists but for database experts. Although existing benchmarks [79] measure performance on different cloud services, extrapolating from those benchmarks to analyzing the performance of a specific database remains a complex undertaking.

More recently, some database services have started to change their mode of interaction with users. Google BigQuery [21], for example, does not support the concept of service instances. Users execute queries and are charged by the gigabyte of data processed. This interface, however, poses different obstacles. It does not offer options to trade-off price and performance (only a choice between "on demand" and "reserved" querying). Furthermore, users have no way to predict performance and for "on demand" querying, the ultimate cost of an analysis.

Research into new SLAs with cloud services has also begun. However, recent proposals require that users have a pre-defined workload [75, 99] or precise query-time constraints [126, 139]. They feature occasionally surprising conditions, such as rejecting queries unless they can execute within the SLA threshold [139].

We argue that no existing approach is satisfactory. Cloud providers should offer a new interface. Instead of showing users the available resources they can lease, or asking users about specific performance requirements for specific workloads, cloud services should show users possible options given their data and let them choose from them. The options should have *performance guarantees* to avoid unexpected behaviors. They should also let users change their price-performance options over time.

This new type of interface raises several challenges. First, for a cloud service to provide performance-based SLAs, it must be able to predict the performance of various queries from the user's dataset. Doing so is complex, especially when the query workload is not well-defined. Second, once a user purchases a performance-based SLA, the system must be able to consolidate resources to ensure that these performance guarantees are met. This presents a cost of the service, not only in terms of instances and storage but for SLA violations. This is especially challenging if there are multiple tenants in the service who have their own SLA.

This thesis explores ways that cloud services can provide interpretable performance-based SLAs without having to pay heavily for SLA violations or over-provisioning to meet the guarantees. To this end, we propose a set of three systems: PSLAManager, SLAOrchestrator (which introduces a new sub-system called PerfEnforce) and DeepQuery. Each focuses on a different aspect of the problem.

1.1 Summary of Contributions and Thesis Outline

Figure 1.2 illustrates these components, which constitute our core contribution, and how they work on top of an existing data analytics service.

First, the system offers a new interface through PSLAManager that balances price-performance goals of users with resource configurations of services (Chapter 2). PSLAManager is an overlay on top of an existing cloud data analytics service. Given a user database uploaded to the cloud, the PSLAManager computes how different service configurations map onto price-performance options for the user database. It provides several options to the user, each with different price and performance trade-offs. Assuming the user has selected an SLA option, Figure 1.2 shows the path of a query; after query q is submitted to the system, and based on the SLA purchased by the user, PSLAManager labels the query with a performance-based SLA, q_{sla}.

Second, to enforce SLAs purchased by multiple users, we present the SLAOrchestrator system (Chapter 3). SLAOrchestrator uses PSLAManager to generate SLAs and introduces a new subsystem, PerfEnforce. Given q_{sla}, PerfEnforce schedules queries and provisions the underlying system. In addition, as it learns more about users' workloads and query execution runtimes for different queries, it improves subsequent PSLAManager SLAs in future sessions.

Third, at the heart of the preceding techniques lies the ability to model and predict query execution times and other properties. These models heavily rely on cost estimates from query optimizers.

Predicting query execution times or other query properties, like cardinality, is extremely difficult. Recently, deep learning (DL) has emerged as a powerful machine learning tool with great success in a variety of areas. In the DeepQuery project (Chapter 4), we empirically evaluate how to use DL to improve the accuracy of cardinality estimation. We focus on the cardinality estimation problem as this is known as the "Achilles Heel" of modern query optimizers [77].

We discuss related work in Chapter 5 and conclude our inquiry in Chapter 6.

We now describe each contribution in more detail.

1.2 PSLAManager: Generating Personalized SLAs in the Cloud

In Chapter 2, we address the fundamental challenge of rethinking the interface between users and the cloud in the context of data analytics. As noted previously, users should be able to select service tiers by thinking only in terms of query performance and query costs, not resource configurations. To do so, we propose the notion of a *Personalized Service Level Agreement (PSLA)*. The key contribution of the PSLA is allowing users to specify the schema of their data and basic statistics (base table cardinalities); the cloud service then shows what types of queries the user can run and the performance of these queries with different levels of service, each with a defined cost. Performance is captured with a maximum query runtime, and queries are represented with templates.

Several challenges arise when generating a PSLA: How many service options should a PSLA have? What types and how many queries should be shown for each option? To help answer these questions, we define metrics to assess the quality of PSLAs. We then develop an approach for generating quality PSLAs from user databases. The PSLAManager appeared at DanaC with SIGMOD'13 [95] and CIDR'15 [96].

1.3 SLAOrchestrator: Reducing the Cost of Performance SLAs for Cloud Data Analytics

In Chapter 3, we address the problem of reducing the primary costs of performance-based SLAs: the cost of resources and the cost of SLA violations. It is difficult for cloud providers to be able to offer affordable performance-based SLAs. They could provide each user with many resources to minimize SLA violations, but doing so would increase the cost of infrastructure. Alternatively, they could provide fewer resources per user, but this could impose costs related to higher risks for SLA violations.

Previous research has addressed the challenge of enforcing SLAs in various ways. One line of work assumes that the SLA is pre-defined [33, 32, 139]. For example, SLA runtimes are artificially generated by offering a performance guarantee 10x the query's true latency [33], or by setting SLAs to the performance of past executions [64]. Without well-defined SLAs, the best enforcement does not help meet performance guarantees. If cloud providers overprovision the underlying system, users have to bear unnecessarily high costs, cloud providers become less competitive, and ultimately users take their business elsewhere. If cloud providers underprovision the system, they must pay penalties for missed SLAs and thus either lose money in the long term or raise prices to compensate.

We develop SLAOrchestrator, a system that enables cloud providers to offer query-level, performance SLAs for data analytics. SLAOrchestrator uses the PSLAManager to show the user options and associated price tags. It generates, updates over time, and enforces SLAs so as to bring the provider's cost down to (or even below) that of the original service without SLAs.

SLAOrchestrator achieves its goal using three key techniques. First, its design is based on the core idea of a double-nested learning loop. In the outer loop, every time a tenant arrives, the system generates a performance SLA given its current model of query execution times. That model learns over time as more tenants use the system. The SLA is in effect for the duration of a *query session*, which we define as the time from the moment users purchase an SLA and issue their first query until they stop their data analysis and exit from the system. In the inner loop, SLAOrchestrator continuously learns from user workloads to improve query scheduling and resource provisioning decisions and reduce costs during query sessions. To drive this inner loop, we introduce a new subsystem, which we call *PerfEnforce*.

Second, the PerfEnforce subsystem represents a new type of query scheduler. Unlike traditional schedulers, which must arbitrate resource access and manage contention, PerfEnforce's scheduler operates in the context of seemingly unbounded, elastic cloud resources. Its goal is *cost-effectiveness*. It schedules queries so as to minimizes over- and under-provisioning overheads. We develop and evaluate four variants of the scheduler. The first is based on a PI controller. Two variants model the problem as either a contextual or non-contextual multi-armed bandit (MAB) [123]. The last uses online learning to model the problem.

Third, PerfEnforce includes a new resource provisioning component. We evaluate two variants of resource provisioning. The first strives to maintain a desired resource utilization level. The second observes tenant query patterns and adjusts both the size of the overall resource pool and the query scheduler's tuning parameters. We present the resource provisioning algorithms in Section 3.3.

We demonstrate that PerfEnforce dramatically reduces service costs for per-query latency SLAs. This work was demonstrated at SIGMOD'16 [97] and published in USENIX ATC '18 [98].

1.4 DeepQuery: An Empirical Analysis of Deep Learning for Cardinality Estimation

In Chapter 4, we focus on using deep learning (DL) as an approach for estimating cardinalities. The SLAOrchestrator system uses scaling techniques that rely heavily on cost estimates provided by the optimizer. In general, inaccurate cost estimates heavily impacts query optimization. Many current optimizers use histograms to estimate cardinalities. For complex queries, optimizers extract information from these histograms in conjunction with "magic constants" to make predictions [78]. Since the estimates from these optimizers are not theoretically grounded, propagating them through each intermediate result of a query plan can cause high cardinality errors, leading to sub-optimal query plans. Although estimating the cost of a query plan is a well-researched problem [115], it remains a challenging one today: existing DBMSs still choose poor execution plans for many queries [78].

To support cardinality estimation, DBMSs collect statistics about the data, which typically take the form of histograms or samples. Because databases contain many tables and columns, these statistics rarely capture all existing correlations. The manual process of selecting the best statistics to collect can help, but it requires significant expertise both in database systems and in the application domain.

Recently, as a result of declining hardware costs and growing datasets available for training, DL has been successfully applied to solving computationally intensive learning tasks in other domains. The advantage of these type of models comes from their ability to learn unique patterns and features of the data [52] that would otherwise be difficult to identify and use for query optimization.

The database community has recently begun to consider the potential of DL techniques to solve database problems [132]. However, there remains a limited understanding of the potential and impact of these models for query optimization. Previous work has demonstrated the potential of using DL as a critical tool for learning indexes [73], improving query plans [87], and learning cardinalities, specifically through deep set models [69]. However, we argue that that the accuracy should not be the only factor to consider when evaluating these models. We must also consider their overheads, robustness, and impact on query plan selection. There is a need for the systematic analysis of the benefits and limitations of various fundamental architectures.

In Chapter 4, we focus on the trade-offs among model size (measured by the number of trainable parameters), the time it takes to train the model, and the accuracy of model predictions. We study these trade-offs for several datasets with the goal of understanding the overheads of these models compared to PostgreSQL's optimizer. To do this, we build several simple neural networks and recurrent neural network models and vary complexity by modifying network widths and depths. We train each model separately and compare model overheads to PostgreSQL and random forest models (based on an off-the-shelf machine learning model).

We find that simple DL models can learn cardinality estimations across a variety of datasets, reducing the error by 72%-98% on average compared to PostgreSQL. In addition, we empirically evaluate the impact of injecting cardinality estimates produced by DL models into the PostgreSQL optimizer. We find that estimates from these models lead to more better query plans across all datasets, reducing runtimes by up to 49%.

A short version of this paper was published at DEEM with SIGMOD'18 [94].

In summary, this thesis addresses how a data analytics service in the cloud can minimize the cost of resources and SLA violations and improve user access to price/performance decision-making. We develop the PSLAManager to generate performance-based SLAs for cloud analytics; the SLAOrchestrator with PerfEnforce and the PSLAManager to drive down the costs of providing performance SLAs; and DeepQuery to offer a new approach to cardinality estimation using deep neural networks.

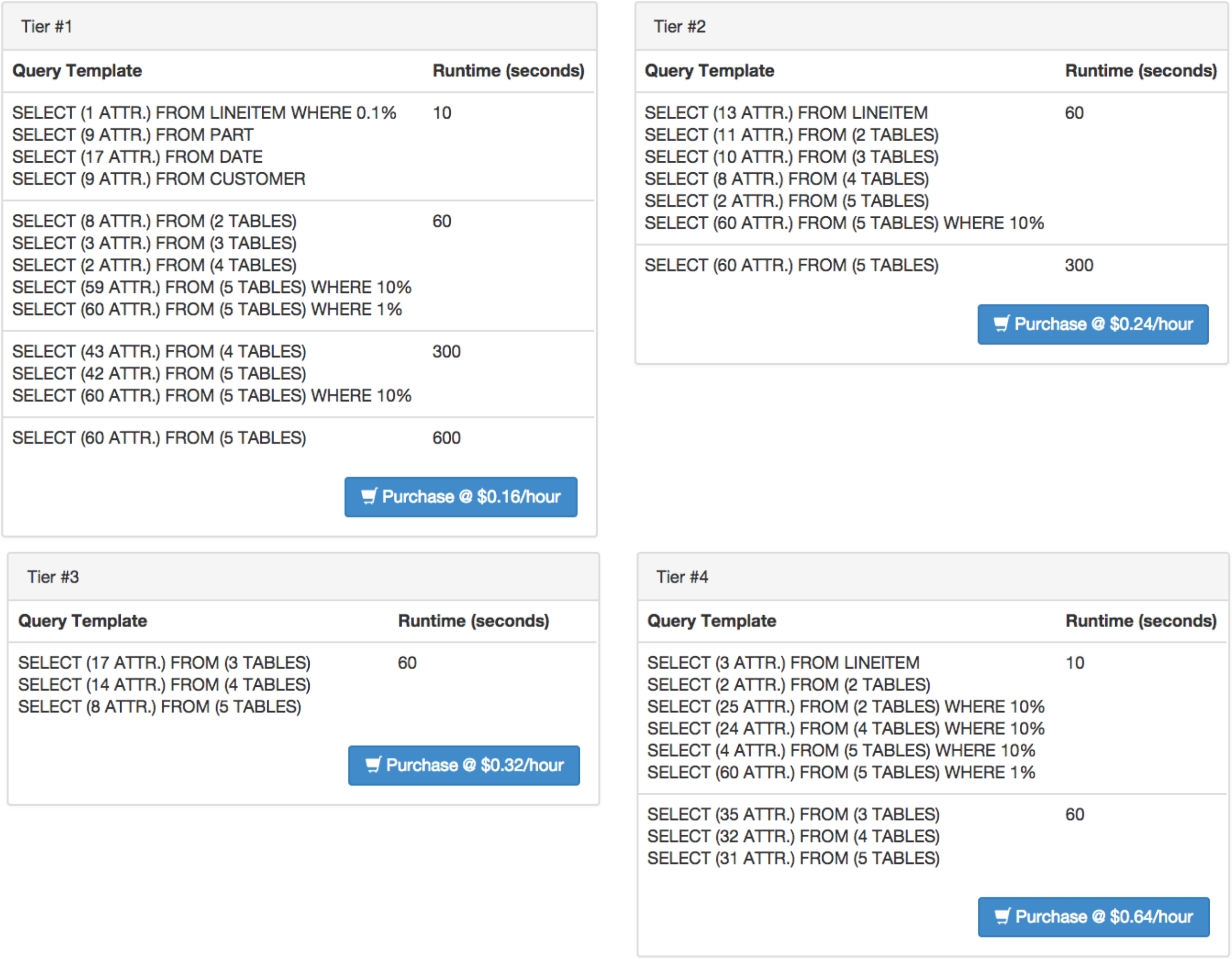
Chapter 2

PSLAMANAGER: GENERATING PERSONALIZED SERVICE LEVEL AGREEMENTS IN THE CLOUD

In this chapter, we focus on the problem of generating performance-based service level agreements for data analytics in the cloud and present our PSLAManager system. This work was originally published at CIDR 2015 [96].

As we described in Section 1.2, our key idea is to introduce a new type of interface to help bridge the gap between the resource-centric offerings shown by cloud providers and what users actually wish to acquire from a cloud service, which is fast query execution performance at a good price. To achieve this goal, we propose the notion of a *Personalized Service Level Agreement (PSLA)*, which is a performance SLA custom-generated for a given user database. We introduce the PSLAManager system, which generates such PSLAs.

The key idea of the PSLA is for a user to specify the schema of her data and basic statistics (base table cardinalities) and for the cloud service to show what types of queries the user can run and the performance of these queries on the user database. Additionally, the PSLA shows query performance for different service levels, each with a defined cost. The performance is captured with a maximum query runtime while queries are represented with templates. Figure 2.1 shows an example PSLA that our service generates (we describe the experimental setup in Section 2.3). The example shows a PSLA with four tiers generated for the Myria shared-nothing DBMS service [57, 2]. The database to analyze is a 10GB instance of the Star Schema Benchmark [93]. The PSLA makes cost and performance trade-offs obvious: If a user needs to execute just a few simple selection queries, Tier 1 likely suffices. These queries will run in under 10sec at that tier. Tier 2 significantly improves the runtimes of the most complex join queries compared with Tier 1. Their runtimes go from below 600sec to below 300sec. Tier 3 provides only limited performance improvement compared with Tier 2. Finally, Tier 4 enables queries with small joins to become interactive with runtimes below 10sec. Figure 2.2 shows a three-tier PSLA for the same dataset but for a single-node SQL Server instance running on Amazon EC2 [12].



Tier #1

Query Template	Runtime (seconds)

SELECT (1 ATTR.) FROM LINEITEM WHERE 0.1%
SELECT (9 ATTR.) FROM PART
SELECT (17 ATTR.) FROM DATE
SELECT (9 ATTR.) FROM CUSTOMER
	10

SELECT (8 ATTR.) FROM (2 TABLES)
SELECT (3 ATTR.) FROM (3 TABLES)
SELECT (2 ATTR.) FROM (4 TABLES)
SELECT (59 ATTR.) FROM (5 TABLES) WHERE 10%
SELECT (60 ATTR.) FROM (5 TABLES) WHERE 1%
	60

SELECT (43 ATTR.) FROM (4 TABLES)
SELECT (42 ATTR.) FROM (5 TABLES)
SELECT (60 ATTR.) FROM (5 TABLES) WHERE 10%
	300

SELECT (60 ATTR.) FROM (5 TABLES)	600

Tier #2

Query Template	Runtime (seconds)

SELECT (13 ATTR.) FROM LINEITEM
SELECT (11 ATTR.) FROM (2 TABLES)
SELECT (10 ATTR.) FROM (3 TABLES)
SELECT (8 ATTR.) FROM (4 TABLES)
SELECT (2 ATTR.) FROM (5 TABLES)
SELECT (60 ATTR.) FROM (5 TABLES) WHERE 10%
	60

SELECT (60 ATTR.) FROM (5 TABLES)	300

Tier #3

Query Template	Runtime (seconds)

SELECT (17 ATTR.) FROM (3 TABLES)
SELECT (14 ATTR.) FROM (4 TABLES)
SELECT (8 ATTR.) FROM (5 TABLES)
	60

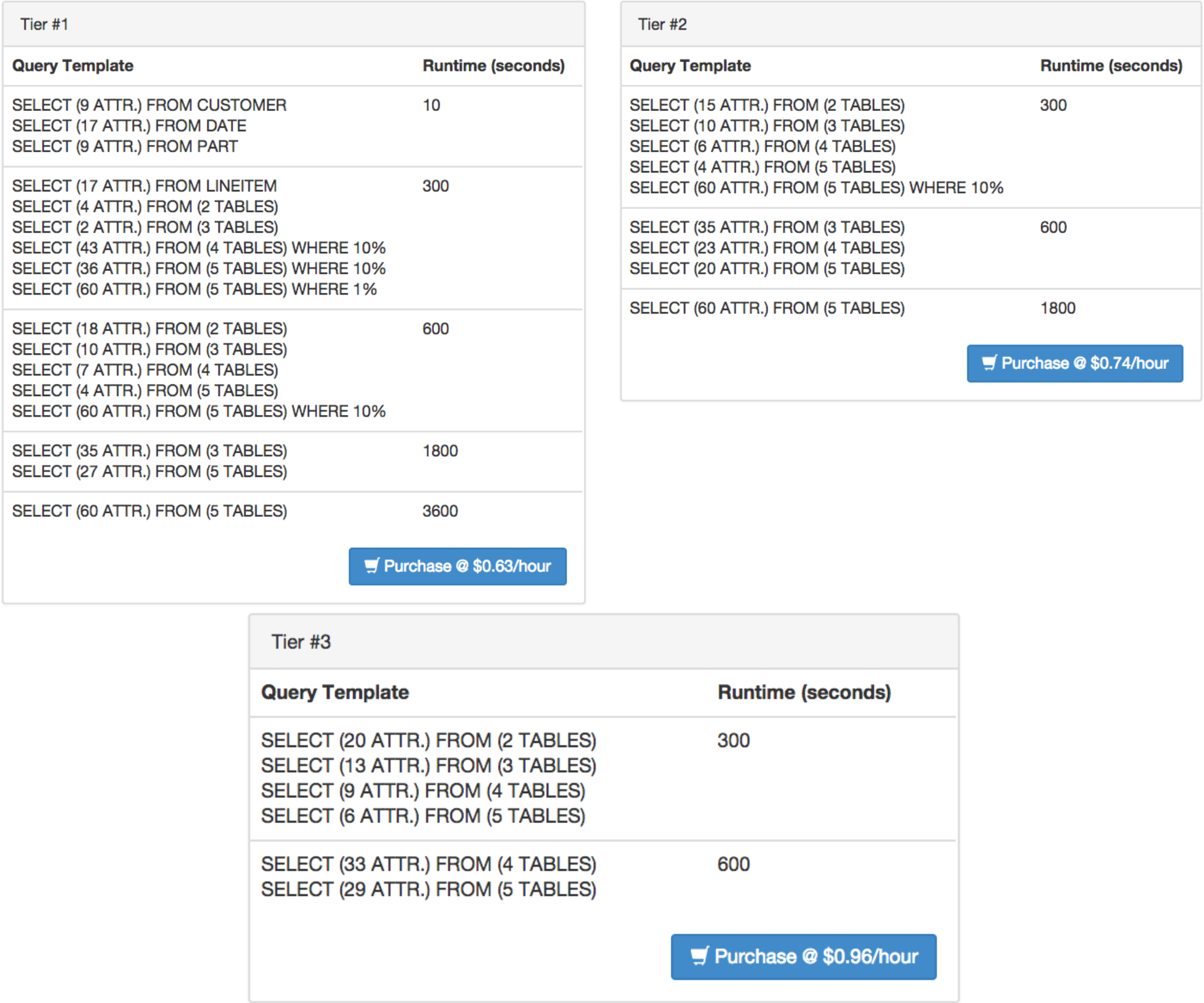
Tier #4

Query Template	Runtime (seconds)

SELECT (3 ATTR.) FROM LINEITEM
SELECT (2 ATTR.) FROM (2 TABLES)
SELECT (25 ATTR.) FROM (2 TABLES) WHERE 10%
SELECT (24 ATTR.) FROM (4 TABLES) WHERE 10%
SELECT (4 ATTR.) FROM (5 TABLES) WHERE 10%
SELECT (60 ATTR.) FROM (5 TABLES) WHERE 1%
	10

SELECT (35 ATTR.) FROM (3 TABLES)
SELECT (32 ATTR.) FROM (4 TABLES)
SELECT (31 ATTR.) FROM (5 TABLES)
	60

Figure 2.1: Example Personalized Service Level Agreement (PSLA) for a 10GB instance of the Star Schema Benchmark on the shared-nothing Myria DBMS service. These four tiers correspond to 4-node, 6-node, 8-node, and 16-node Myria deployments.



Tier #1

Query Template	Runtime (seconds)

SELECT (9 ATTR.) FROM CUSTOMER
SELECT (17 ATTR.) FROM DATE
SELECT (9 ATTR.) FROM PART
	10

SELECT (17 ATTR.) FROM LINEITEM
SELECT (4 ATTR.) FROM (2 TABLES)
SELECT (2 ATTR.) FROM (3 TABLES)
SELECT (43 ATTR.) FROM (4 TABLES) WHERE 10%
SELECT (36 ATTR.) FROM (5 TABLES) WHERE 10%
SELECT (60 ATTR.) FROM (5 TABLES) WHERE 1%
	300

SELECT (18 ATTR.) FROM (2 TABLES)
SELECT (10 ATTR.) FROM (3 TABLES)
SELECT (7 ATTR.) FROM (4 TABLES)
SELECT (4 ATTR.) FROM (5 TABLES)
SELECT (60 ATTR.) FROM (5 TABLES) WHERE 10%
	600

SELECT (35 ATTR.) FROM (3 TABLES)
SELECT (27 ATTR.) FROM (5 TABLES)
	1800

SELECT (60 ATTR.) FROM (5 TABLES)	3600

Tier #2

Query Template	Runtime (seconds)

SELECT (15 ATTR.) FROM (2 TABLES)
SELECT (10 ATTR.) FROM (3 TABLES)
SELECT (6 ATTR.) FROM (4 TABLES)
SELECT (4 ATTR.) FROM (5 TABLES)
SELECT (60 ATTR.) FROM (5 TABLES) WHERE 10%
	300

SELECT (35 ATTR.) FROM (3 TABLES)
SELECT (23 ATTR.) FROM (4 TABLES)
SELECT (20 ATTR.) FROM (5 TABLES)
	600

SELECT (60 ATTR.) FROM (5 TABLES)	1800

Tier #3

Query Template	Runtime (seconds)

SELECT (20 ATTR.) FROM (2 TABLES)
SELECT (13 ATTR.) FROM (3 TABLES)
SELECT (9 ATTR.) FROM (4 TABLES)
SELECT (6 ATTR.) FROM (5 TABLES)
	300

SELECT (33 ATTR.) FROM (4 TABLES)
SELECT (29 ATTR.) FROM (5 TABLES)
	600

Figure 2.2: Example Personalized Service Level Agreement (PSLA) for a 10GB instance of the Star Schema Benchmark on a single-node Amazon EC2 instance with SQL Server. The three tiers correspond to a small, medium, and large EC2 instance.

In this chapter, we present the *PSLAManager*, a new system for the generation and management of PSLAs. The challenge behind developing the PSLAManager lies in developing a model to reason about PSLA quality and algorithms to generate high-quality PSLAs. We tackle those two challenges in this chapter. More specifically, we make the following contributions:

- In Section 2.1, we define a model for PSLAs and metrics to assess PSLA quality. These metrics include performance error (how accurately the displayed time-thresholds match the actual query times), complexity (a measure of the PSLA size), and query capabilities (the types of queries described in the PSLA).
- In Section 2.2, we develop a method to automatically generate a PSLA for a cloud service and user database. The challenge is to generate PSLAs with low performance error and low complexity at the same time while preserving a given set of query capabilities.
- In Section 2.3, we show experimentally, using both Amazon EC2 [12] and our Myria parallel data management service [57], that our approach can generate PSLAs with low errors and low complexity.

The PSLAManager generates a PSLA for a given database and cloud service. This system can thus be layered on top of an existing cloud data management service such as Amazon RDS, Amazon Elastic MapReduce, or equivalent. In this chapter, we assume such a cloud-specific deployment that gives the PSLAManager access to the cloud service internals including the query optimizer (which we use to collect features of query plans for runtime predictions). However, since the PSLAManager takes as input only the user's database schema and statistics, it could also be a middleware service that spans multiple clouds and facilitates the comparison of each service's price-performance trade-offs through the common PSLA abstraction. The two PSLAs shown in Figures 2.1 and 2.2, for example, facilitate the comparison of the Amazon EC2 with SQL Server service and the Myria service given the user's database.

2.1 PSLA Model

We first define a Personalized Service Level Agreement (PSLA) more precisely together with quality metrics for PSLAs.

We start by defining the notion of a *query template*. The term "query template" has traditionally been used to refer to parameterized queries. Rajaraman et al. [109] used the term query template to designate parameterized queries that differ in their selection predicates. Agarwal et al. [6], on the other hand, refer to queries with different projected attributes. We generalize the notion of a query template to select-project-join (SPJ) queries that differ in the projected attributes, relations used in joins (the joined tables are also parameters in our templates), and selection predicates.

Query Template: A *query template* M for a database D is a triple $M = (F, S, W)$ that compactly represents several possible SPJ queries over D: F represents the maximum number of tables in the FROM clause. S represents the maximum number of projected attributes in the SELECT clause. W represents the maximum overall query selectivity (as a percent value), based on the predicate in the WHERE clause. Joins are implicitly represented by PK/FK constraints. No cartesian products are allowed.

For example, the template (4, 12, 10%), represents all queries that join *up to* four tables, project up to 12 attributes, and select up to 10% of the base data in the four tables.

We can now define a Personalized Service Level Agreement:

Personalized Service Level Agreement: A PSLA for a cloud provider C and user database D is a set of PSLA tiers for the user to choose from, , $PSLA(C, D) = \{R_1, R_2, \ldots, R_k\}$, where each tier, R_i is defined as:

$$\begin{aligned} R_i = (p_i, d_i, \{&(th_{i1}, \{M_{i11}, M_{i12}, \ldots, M_{i1q_a}\}), \\ &(th_{i2}, \{M_{i21}, M_{i22}, \ldots, M_{i2q_b}\}), \\ &\ldots, \\ &(th_{ir}, \{M_{ir1}, M_{ir2}, \ldots, M_{irq_c}\})\}) \end{aligned}$$

Each tier has a fixed hourly price, p_i, and no two tiers have the same price. Each tier also has a set of query templates M_{i11} through M_{irq_c} clustered into r groups. Each of these templates is unique within each tier. Each template group, j, is associated with a time thresholds th_{ij}. Finally, d_i is the penalty that the cloud agrees to pay the user if a query fails to complete within its corresponding time threshold th_{ij}.

Figures 2.1 and 2.2 show two example PSLAs for the Star Schema Benchmark [93] dataset. One PSLA is for the Myria shared-nothing cloud service [57] while the other is for the Amazon EC2 service with SQL Server [12] (we describe the experimental setup in Section 2.3). The figures shows screenshots of the PSLAs as they are shown in our PSLAManager system. To display each template $M = (F, S, W)$, we convert it into a more easily readable SQL-like format as shown in the figures.

Given a PSLA for a specific cloud service, the user will select a service tier R_i and will be charged the corresponding price per time unit p_i. The user can then execute queries that follow the templates shown and is guaranteed that all queries complete within the specified time-thresholds (or the cloud provider incurs a penalty d_i). Query runtimes across tiers either decrease or stay approximately constant depending on whether the selected configurations (or tiers) improve performance. If some queries take a similar amount of time to process at two tiers of service R_j and $R_i, j < i$, the queries are shown only for the cheaper tier, R_j: a query that can be expressed through a template from a lower tier, R_j, but has no representative template in the selected tier, R_i, will execute with the expected runtime shown for the lower tier R_j.

There are several challenges related to generating a PSLA: How many tiers should a PSLA have? How many clusters and time-thresholds should there be? How complex should the templates get? To help guide the answers to these questions, we define three metrics to assess the quality of a PSLA:

PSLA Complexity Metric: The complexity of a PSLA is measured by the number of query templates that exist in the PSLA for all tiers.

The intuition for the complexity metric is that, ultimately, the size of the PSLA is determined by the number of query templates shown to the user. The smaller the PSLA complexity, the easier it is for the user to understand what is being offered. Hence, we prefer PSLAs that have fewer templates and thus, a lower complexity.

PSLA Performance Error Metric: The PSLA performance error is measured as the root mean squared error (RMSE) between the estimated query runtimes for queries that can be expressed following the PSLA templates in a cluster and the time-thresholds associated with the corresponding cluster. The RMSE is first computed for each cluster. The PSLA error is the average error across all clusters in all tiers.

The RMSE computation associates each query with the template that has the lowest time threshold and that can serve to express the query.

The intuition behind the above error metric is that we want the time thresholds shown in PSLAs to be as close as possible to the actual query runtimes. Minimizing the RMSE means more compact query template clusters, and a smaller difference between query runtime estimates and the time-thresholds presented to the user in the PSLA.

Given a query-template cluster containing queries with expected runtimes $\{q_1, \ldots, q_k\}$, and time-threshold th, the RMSE of the cluster is given by the following equation. Notice that we use *relative runtimes* because queries can differ in runtimes by multiple orders of magnitude.

$$RMSE(\{q_1, \ldots, q_k\}, th) = \sqrt{\frac{1}{k}\sum_{i=1}^{k}\left(\frac{q_i - th}{th}\right)^2}$$

Because the set of all possible queries that can be expressed following a template is large, in our system, we measure the RMSE using only the queries in the workload that our approach generates.

PSLA Capability Coverage: The class of queries represented by the query templates: , selection queries, selection queries with aggregation, full conjunctive queries, conjunctive queries with projection (SPJ queries), conjunctive queries with aggregation, union of conjunctive queries, queries with negations.

For this last metric, higher capability coverage is better since it provides users with performance guarantees for more complex queries over their data.

Problem Statement: Given a database D and a cloud DBMS C, the problem is how to gen-

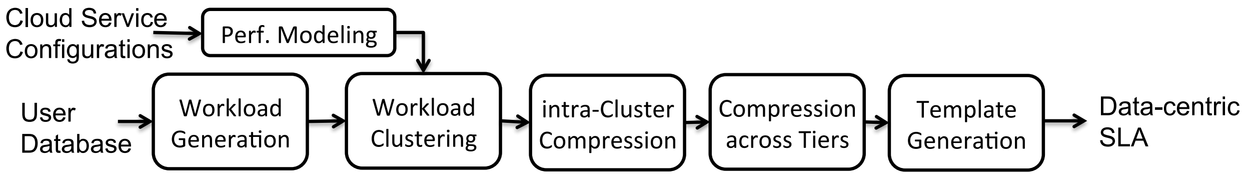


Figure 2.3: PSLA generation process.

erate a PSLA comprising a set of PSLA tiers R that have low complexity, low performance error, and high capability coverage. The challenge is that complexity, performance and capability coverage are at odds with each other. For instance, the higher we extend the capability coverage for the queries, the higher the PSLA complexity becomes and the likelihood of performance error increases as well. Additionally, these PSLAs must be generated for a new database each time, or any time the current database is updated. Hence, PSLA generation should be fast.

2.2 PSLA Generation

We present our approach for generating a PSLA given a database D and a cloud service C. We keep the set of query capabilities fixed and equal to the set of SPJ queries. However, there is no fundamental limitation that prevents extending the approach to more complex query shapes. Figure 2.3 shows the overall workflow of the PSLA generation process. We present each step in turn.

We focus on databases D that follow a star schema: a fact table f and a set of dimension tables $\{d_1, \ldots, d_k\}$. Queries join the fact table with one or more dimension tables. Star schemas are common in traditional OLAP systems. They are also common in modern analytic applications: A recent paper [6], which analyzed queries at Facebook, found that the most common type of join queries were joins between a single large fact table and smaller dimension tables. It is possible to extend our algorithms to other database schemas, as long as the PK/FK constraints are declared, so that the system can infer the possible joins.

2.2.1 *Tier Definition*

The first question that arises when generating a PSLA is what should constitute a service tier? Our approach is to continue to tie service tiers to resource configurations because, ultimately, resources are limited and must be shared across users. For example, for Amazon EC2, three natural service tiers correspond to a small, medium, and large instance of that service. For Amazon Elastic MapReduce, possible service tiers are different-size clusters (, cluster of size 2, 3, 4, . . ., 20). We leave it to the cloud provider to set possible resource configurations. Importantly, the cloud can define a large number of possible configurations. The PSLAManager filters-out uninteresting configurations during the PSLA generation process as we describe in Section 2.2.4. For example, in the case of a shared-nothing cluster, the provider can specify that each cluster size from 1 to 20 is a possible configuration. Our PSLAManager will pick a subset of these configurations to show as service tiers.

The mapping from service tiers to resources is invisible to users. In recent work, it has been shown that cloud providers can often choose different concrete resource configurations to achieve the same application-level performance [62]. For now, we define each service tier to map onto a specific resource configuration.

2.2.2 *Workload Generation*

The first step of PSLA generation is the production of a workload of queries based on the user's database. Recall that, in our approach, we do not assume that the user already has a workload. Instead, we only require that the user provides as input a schema and basic statistics on their data. The PSLAManager generates the PSLA from this schema and statistics.

The goal of the PSLA is to show a distribution of query runtimes on the user data. Hence, fundamentally, we want to generate valid queries on the user schema and estimate their runtime for different resource configurations. The key question is, what queries should be included in this process? If we generate too many queries, PSLA generation will become slow and the PSLAs may become complex. If we generate too few queries, they will not sufficiently illustrate performance trade-offs.

From experience working with different cloud DBMS services, we find [95] that, as can be expected, some of the key performance differences revolve around join processing. Therefore, our approach is to focus on generating queries that illustrate various possible joins based on the user's data. To avoid generating all possible permutations of tables, we apply the following heuristic: For each pattern of table joins, we generate the query that will process the largest amount of data. Hence, our algorithm starts from considering each table in isolation. It then considers the fact table joined with the largest dimension table. Next, it generates queries that join the fact table with the two largest dimension tables, and so on until all tables have been joined. The goal is to generate one of the expensive queries among equivalent queries to compute time thresholds that the cloud can more easily guarantee.

Of course, users typically need to look at subsets of the data. Changing the query selectivity can significantly impact the size and thus performance of the joins. To show these trade-offs, we add to each of the above queries selection predicates that retain different orders of magnitude of data such as 0.1%, 1%, 10%, and 100%. To ensure that the predicates change the scales of the join operations, we apply them on the primary key of the fact table.

Finally, since many systems are column-stores, we generate queries that project varying numbers of attributes.

Algorithm 1 shows the pseudocode for the query generation step. The input is the database $D = \big\{\{f\} \cup \{d_1, \ldots, d_k\}\big\}$, each table $t_i \in D$ has a set of attributes $A(t_i)$. Function $pk(t_i)$ returns the set of primary key attributes for t_i. The fact table f has a foreign key for each dimension table, that references the dimension table's primary key. The algorithm generates the set of representative queries Q given the database D. The result set Q consists of triples, (T^q, A^q, e^q), where T^q is the set of tables in the FROM clause, A^q is the set of projected attributes in the SELECT clause, and e^q is the desired selectivity value, which translates into a predicate on the PK of the fact table in the WHERE clause (or predicate on the PK of a dimension table if there is no fact table in the FROM clause). The algorithm keeps a list L with all representative sets of tables T^q. Initially every standalone table $t_i \in D$ is inserted as a singleton set into L (lines 4-6), then the fact table is

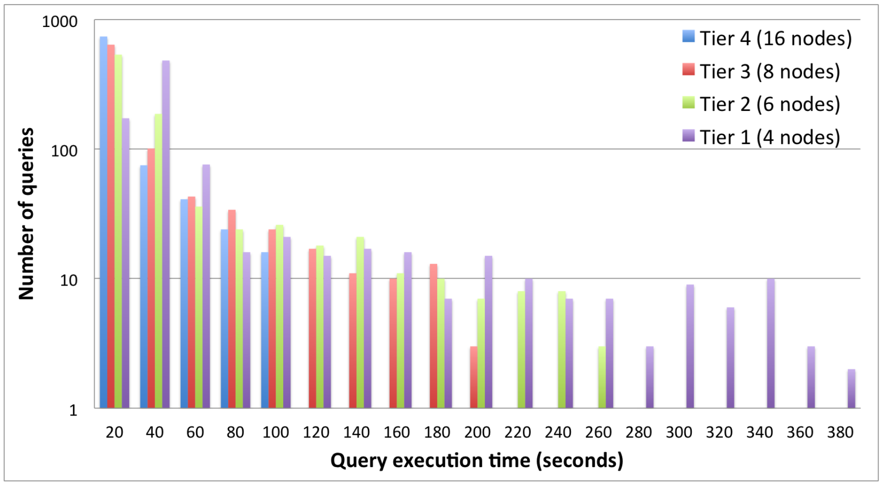
Algorithm 1 Query generation algorithm

```
1: Input: $D$
2: $Q \leftarrow \{\}, L \leftarrow \{\}$
3: // Step 1: Generate combinations of tables for the FROM clause
4: for each $t_i \in D$ do
5:     $T^q \leftarrow \{t_i\}$
6:     $L \leftarrow L \cup T^q$
7:     // For the fact table add combinations of dimensions tables
8:     if $t_i$ is the fact table, i.e. $t_i = f$ then
9:         Sort $\{d_1, \ldots, d_k\}$ desc. on $size(d_i)$, $1 \leq i \leq k$
10:        for each $j$, $1 \leq j \leq k$ do
11:            $D^q \leftarrow$ take the first $j$ tables from $D$
12:            $T^q \leftarrow T^q \cup D^q$
13:            $L \leftarrow L \cup T^q$
14: // Step 2: Add the projections and selections
15: for each $T^q \in L$ do
16:     Sort $A(T^q)$ desc. on $size(A_j(T^q))$, $1 \leq j \leq |A(T^q)|$
17:     for each $k$, $1 \leq k \leq |A(T^q)|$ do
18:         // Project the $k$ largest attributes
19:         $A^q \leftarrow$ take the first $k$ attributes from $A(T^q)$
20:         // Add a desired selectivity from a pre-defined set
21:         for each $e^q \in E_{T^q}$ do
22:             $Q \leftarrow Q \cup \{T^q, A^q, e^q\}$
    return $Q$
```

expanded (line 8), when we generate the joins.

2.2.3 *Query Time Prediction*

Given the generated workload, a key building block for our PSLA approach is the ability to estimate query runtimes. Here, we build on extensive prior work [49, 48, 7] and adopt a method based on machine learning: Given a query, we use the cloud service's query optimizer to collect query plan features including the estimated query result size and total query plan cost among other features. We build a model offline using a separate star schema dataset. We use that model to predict the runtime of each query in the generated workload given its feature vector and a resource configuration. With this approach, to generate a PSLA, users only need to provide the schema of their database and basic statistics such as the cardinality of each input relation.



Bucket size	**EMD(1,2)**	**EMD(2,3)**	**EMD(3,4)**	**EMD(2,4)**
5	17.53	8.70	12.00	20.70
10	17.43	7.07	13.74	20.81
20	20.58	7.83	9.91	17.75

Figure 2.4: Distribution of query times across the four initially considered configurations of Myria. The PSLAManager automatically identifies the tiers with the most different query runtime distributions by computing the Earth Mover's Distance (EMD) between tiers. EMD(i,j) is the EMD between service tiers i and j.

2.2.4 Tier Selection

Once our system generates a workload and estimates query runtimes for each resource configuration defined by the cloud provider, our approach is to select a small number of these resource configurations to show as service tiers. Figure 2.4 illustrates the approach using real query runtime distributions obtained for the 10GB SSB dataset and the Myria service. The figure shows the query runtime distributions (plotted as a histogram with buckets of size 20 sec) for four cluster sizes (4, 6, 8, and 16 nodes). As the figure shows, and as expected, the query runtime distribution shifts to the left with larger Myria clusters. The goal is to narrow down which service tiers to show to the user.

Because the goal of the PSLA is to show users different points in the space of price-performance trade-offs, the PSLAManager selects the configurations that differ the most in the distributions of estimated query runtimes for the given workload. To compute these distances, we use the Earth Mover's Distance [112], since this method effectively compares entire data distributions; in our case, the distribution of query runtimes.

The tier selection algorithm proceeds as follows: Given a sequence of increasingly costly resource configurations: $c_1, c_2, \ldots, c_k$, we compute the distances $EMD(c_i, c_{i+1}) \forall i \in [1, k)$. We then recursively select the smallest distance in the list and remove the more expensive configuration c_{i+1}, and recompute the distance between the new neighbors $EMD(c_i, c_{i+2})$, if c_{i+2} exists. We terminate once we have a desired number $k' < k$ of tiers. Figure 2.4 shows the EMD values for different histogram granularities for the Myria example. Assuming a distribution captured with a fine-grained histogram with buckets of size 5sec or 10sec, and assuming the goal is to show only two tiers, the algorithm first eliminates the third configuration because EMD(2,3) is the smallest EMD value. It then re-computes EMD(2,4). At this point, EMD(1,2) is smaller than EMD(2,4). As a result, the second configuration is discarded. The final two tiers selected are tier 1 and 4. They correspond to the 4-node and 16-node configurations (we further discuss this figure in Section 2.3). Observe that the granularity with which the query time distributions are captured can affect the choice of service tiers. In this example, if we use buckets of size 20sec, the final two tiers selected are Tiers 1 and 2.

2.2.5 Workload Compression into Clusters

Our workload-generation approach reduces the set of all possible queries down to a set of representative queries, Q. This set may still easily range in the hundreds of queries as in the case of the SSB schema, for which our approach produces 896 queries.

It would be overwhelming to show the entire list of queries from the generated workload to the user. This would yield PSLAs with low error but high complexity as defined in Section 2.1. Instead, our approach is to compress the representative workload. The compression should reduce the PSLA complexity, which is measured by the number of query templates, while keeping

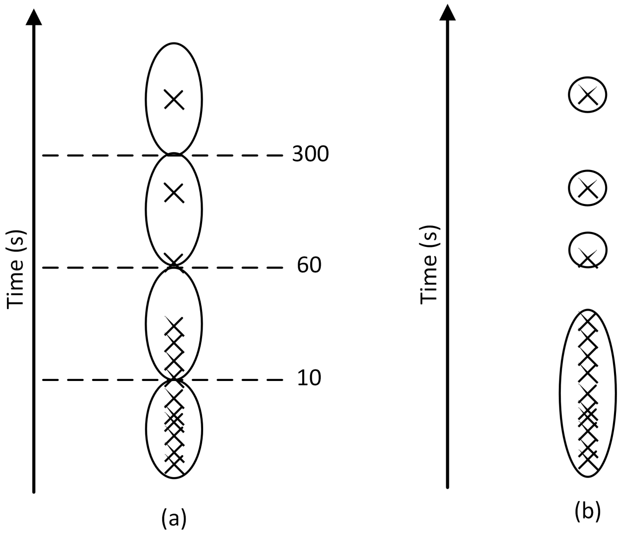


Figure 2.5: Clustering techniques illustration. Each X corresponds to a query. (a) Threshold-based clustering (b) Density-based clustering.

the performance error low. These two optimization goals are subject to the constraint that the compression should preserve capability coverage from the original workload: For example, if the original workload includes a query that joins two specific tables, the final query templates should allow the expression of such a query.

Our workload compression approach has three components as shown in Figure 2.3. The first component is query clustering. We start with the cheapest tier and use a clustering technique to group queries based on their runtimes in that tier. We consider two different types of clustering algorithms: threshold-based and density-based. Figure 2.5 illustrates these two methods.

For threshold-based clustering, we pre-define a set of thresholds and partition queries using these thresholds. The PSLAs shown in Figure 2.1 and Figure 2.2 result form this type of clustering. We use two approaches to set the thresholds: (i) we vary the thresholds in fixed steps of 10, 100, 300, 500, and 1000 seconds, which we call INTERVAL10, INTERVAL100, INTERVAL300, INTERVAL500, and INTERVAL1000, respectively; and (ii) we vary the thresholds following different logarithmic scales. One of the scales is based on powers of 10, which we call LOG10. The other one is based on human-oriented time thresholds of 10sec, 1min, 5min, 10min, 30min, and

1hour, which we call LOGHUMAN. These intervals are represented by the dashed horizontal lines as seen in Figure 2.5(a). The benefit of this approach is that the same time-thresholds can serve for all tiers, making them more comparable. However, the drawback is that it imposes cluster boundaries without considering the distribution of the query runtimes.

For density-based clustering, we discover clusters of queries within a span of a given amount of seconds. We explore varying the parameter (ε) of the DBSCAN algorithm, in order to discover clusters of queries within a span of 10, 100, 300, 500, and 1000 seconds, which we call DBSCAN10, DBSCAN100, DBSCAN300, DBSCAN500, and DBSCAN1000, respectively. Specifically, we begin with a small ε value that produces small clusters throughout a tier. We iterate and slowly increment this value until clusters of the specified size are found. We evaluate and discuss these different clustering algorithms in Section 2.3.

An example of clusters using the threshold-based and density-based approaches are shown in Figure 2.5 (a) and (b), respectively. As the figure shows, the choice of the interval $[0-10\text{sec}]$ breaks an obvious cluster of queries into two. On the other hand, in the $[1-5\text{min}]$ interval, DBSCAN, given the density parameters used, creates two different singleton clusters for two queries inside this interval, which could arguably be combined into one cluster.

For each tier, the generated clusters determine the time-thresholds, th_i, that will be shown in the resulting PSLA for the corresponding tier.

2.2.6 *Template Extraction*

Once we cluster queries, we compute the smallest set of templates that suffice to express all queries in each cluster. We do so by computing the skyline of queries in each cluster that *dominate* others in terms of query capabilities.

Since the queries that our approach generates vary in the tables they join, the attributes they project, and their overall selectivity, we define the dominance relationship ($\sqsupseteq$) between two queries $q_i = (T_i, A_i, e_i)$ and $q_j = (T_j, A_j, e_j)$ as follows:

$$q_i \sqsupseteq q_j \iff T_i \sqsupseteq T_j \land A_i \sqsupseteq A_j \land e_i \supseteq e_j$$

which can be read as q_i dominates q_j iff the set of tables of q_i (T_i) dominates the set of tables of q_j (T_j), the set of attributes of q_i (A_i) dominates the set of attributes of q_j (A_j), and the selectivity of q_i (e_i), dominates the selectivity of q_j (e_j). We say that the set of tables T_i dominates T_j iff the number of tables in T_i is larger than in T_j. Similarly, a set of attributes A_i dominates another set A_j iff the number of attributes in A_i is larger than in A_j. For selectivities, we simply check whether e_i is greater than e_j.

We call each query on the skyline a *root query*. Given a cluster, for each root query q, we generate a template $M = (F, S, W)$, where F is the number of tables in q's FROM clause, S is the number of attribute in q's SELECT clause, and W is the percent value of q' selectivity. For single-table queries, we keep the table name in the template (no table dominance). We do this to help distinguish between the facts table and a dimension table.

Tables 2.1 and 2.2 in Section 2.3 show the reduction in the number of queries shown in PSLAs thanks to the query clustering and template extraction steps.

2.2.7 Cross-Tier Compression

Once we generate the PSLA for one tier, we move to the next, more expensive tier. We observe that some queries have similar runtimes across tiers.

As we indicated in Section 2.1, by choosing one tier, a user gets the level of service of all lower tiers, plus improvements. Hence, if some queries do not improve in performance across tiers, the corresponding templates should only be shown in the cheapest tier. Recall that we measure PSLA complexity by counting the total number of query templates. Hence showing fewer templates improves that quality metric.

Hence, to reduce the complexity of PSLAs, we drop queries from the more expensive tier if their runtimes do not improve compared with the previous, cheaper tier. We call this process *cross-tier compression*. More precisely, query q_j is removed from a tier R_j if there exists a query q_i in tier R_i with $i < j$ such that $q_i \sqsupseteq q_j$ and q_j's runtime in R_j falls within the runtime of q_i's cluster in R_i. We check for dominance instead of finding the exact same query because that query could have previously been removed from the cheaper tier. Tables 2.1 and 2.2 in Section 2.3 show the benefit of this step in terms of reducing the PSLA complexity.

We run the clustering algorithm for the new tier only after the cross-tier compression step.

We also experimented with merging entire clusters between tiers. This approach first computes the query clusters separately for each tier. It then removes clusters from more expensive tiers by merging them with similar clusters in less expensive tiers. A cluster from a more expensive tier is merged only when all of its containing queries are dominated by queries in the corresponding cluster in the less expensive tier. This approach, however, resulted in a much smaller opportunity for cross-tier compression than the per-query method and we abandoned it.

2.3 Evaluation

We implement the PSLAManager in C# and run it on a 64-bit Windows 7 machine with 8GB of RAM and an Intel i7 3.40GHz CPU. We use the WEKA [56] implementation of the M5Rules [107] technique for query runtime predictions.

We evaluate the PSLAManager approach using two cloud data management services: Amazon EC2 [12] running an instance of SQL Server and our own Myria service [2], which is a shared-nothing parallel data management system running in our private cluster. For Amazon, we use the following EC2 instances as service tiers: Small (64-bit, 1 ECU, 1.7 GB Memory, Low Network), Medium (64-bit, 2 ECU, 3.75 GB Memory, Moderate Network), and Large (64-bit, 4 ECU, 7.5 GB Memory, Moderate Network). We use the SQL Server Express Edition 2012 provided by each of the machines through an Amazon Machine Image (AMI). For Myria, we use deployments of up to 16 Myria worker processes (, nodes) spread across 16 physical machines (Ubuntu 13.04 with 4 disks, 64 GB of RAM and 4 Intel(R) Xeon(R) 2.00 GHz processors). We evaluate the PSLA generation technique across 4 different configurations (4, 6, 8, and 16 nodes) for Myria.

To build the machine learning model to predict query runtimes, we first generate a synthetic dataset using the Parallel Data Generation Framework tool [108]. We use this tool to generate a 10GB dataset that follows a star schema. We call it the PDGF dataset. It includes one fact table and five dimensions tables of different degrees and cardinalities. The PDGF dataset contains a total of 61 attributes. For our testing dataset, we use a 10GB database generated from the TPC-H

Star Schema Benchmark (SSB) [93] with one fact table, four dimension tables and a total of 58 attributes. We choose this dataset size because multiple Hadoop measurement papers report 10GB as a median input dataset analyzed by users today [110].

Since Myria is running in our private cloud, to price the service, we use the prices of the same cluster sizes for the Amazon EMR service [13]. For Amazon, we use prices from single-node Amazon EC2 instances that come with SQL Server installed.

2.3.1 Concrete PSLAs

We start by looking at the concrete PSLAs that the PSLAManager generates for the SSB dataset and the Amazon and Myria cloud services. Figures 2.1 and 2.2 show these PSLAs. The PSLAs use the LOGHUMAN threshold-based clustering method (Section 2.2.5). We use real query runtimes when generating these PSLAs. We discuss query time prediction errors and their impact on PSLAs in Sections 2.3.6 and 2.3.7, where we also show the PSLAs produced when using predicted times (see Figures 2.9 and 2.10).

As Figure 2.1 shows, the PSLA generated for Myria has a low complexity. Each tier has only between 3 and 13 templates grouped into four clusters or less. The PSLA has a total of 32 templates. These few templates represent all SPJ queries for this database (without self joins). The PSLA also has a low average RMSE of 0.20. Most of the error stems from the query distribution in the cheapest tier under threshold 600: All workload queries in that tier and cluster have runtimes below 371sec.

Each service tier corresponds to one of the four Myria cluster configurations, but users need not worry about these resource configurations. The price and performance trade-offs are clear: If the user plans to run a few simple selection queries, then Tier 1 suffices. These queries will already run in under 10sec at that tier. If the user plans to perform small joins (joining few tables or small subsets of tables), Tier 4 can run such queries at interactive speed, below 10sec. In contrast, if the user will mostly run complex join queries on the entire dataset, then Tier 2 is most cost effective.

Interestingly, the figure also shows that some tiers are more useful than others. For example, we can see that Tier 3 improves performance only marginally compared with Tier 2. In Section 2.3.3, we show that the PSLAManager will drop that tier first if requested to generate a 3-tier PSLA.

Figure 2.2 shows the three-tier PSLA generated for the SSB dataset and the three single-node Amazon instances. This PSLA has more clusters (time thresholds) than the Myria PSLA due to the wider spread in query runtime distributions: 10, 300, 600, 1800, and 3600. Each tier ultimately has between 6 and 17 templates, which is higher than in the Myria PSLA. The total number of templates, however, is interestingly the same. The error for this PSLA is slightly higher with an RMSE of 0.22. Similarly to the Myria PSLA, most of the error comes from the cheapest tier, where more queries happen to fall toward the bottom of the clusters.

Interestingly, the PSLAs not only make the price-performance trade-offs clear for different resource configurations of the same cloud service, they also make cloud services easier to compare. In this example, the PSLAs clearly show that the Myria service is significantly more cost-effective than the single-node Amazon service for this specific workload and database instance.

Next, we evaluate the different components of the PSLA-generation process individually. We first evaluate the different steps using real query runtimes. We generate the query workload using the algorithm from Section 2.2.2. For Amazon, we execute all queries from the SSB dataset three times and use the median runtimes to represent the real runtimes. For Myria, we execute all queries for the same dataset only once. In this case, the runtimes have little variance as we run each query on a dedicated cluster and a cold cache. In Section 2.3.6, we study the query time variance and query time predictions. We examine the impact of using predicted times on the generated PSLAs compared to real runtimes in Section 2.3.7.

2.3.2 Workload Generation

Our workload-generation algorithm systematically enumerates specific combinations of SPJ queries and, for each combination, outputs one representative query. With this approach, the workload remains small: , we generate only 896 queries for the SSB benchmark. However, this approach assumes that the selected query represents the runtime for other similar queries.

We evaluate the quality of the representative queries that our generator selects by randomly generating 100 queries on the SSB benchmark and associating them with the workload query that

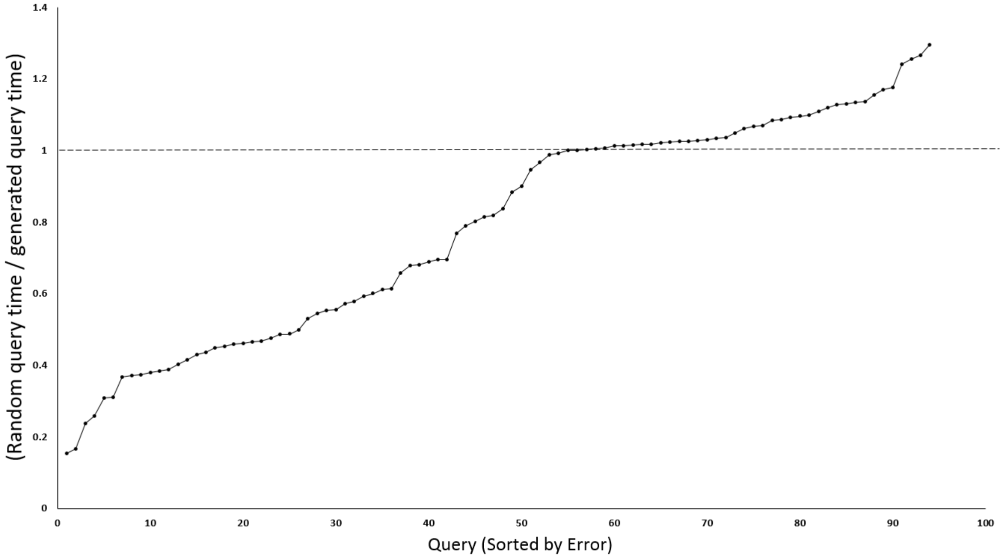


Figure 2.6: Runtimes of queries in the generated workload Q compared with 100 random queries on the medium Amazon instance.

is its representative. The representative query should be more expensive to process. We execute the random queries on the Medium EC2 instance and find that he PSLAManager has either a faster runtime or a similar runtime (within 20%) for 90% of the queries. We show these results in Figure 2.6. This shows that the generated queries are indeed a good representative of the expected upper-bound on the runtime for the user query. The remaining 10% of queries show a limitation of our approach and the need for the cloud to have techniques in place to counteract unexpected query runtimes.

2.3.3 Tier Selection

Given a workload of queries with associated query runtimes, the next step of PSLA generation is tier selection. To select tiers, the PSLAManager takes the distribution of query runtimes for all tiers and computes the earth mover's distance (EMD) between them. It then eliminates the tiers with the most similar EMD values.

We evaluate this approach for both services. In our experiments, the Myria service has four configurations that correspond to clusters of sizes 4, 6, 8, and 16. To capture the query time distributions, we partition the runtimes into buckets of size 5, 10, or 20. We then compute the earth mover's distance between all consecutive pairs of tiers. Figure 2.4 shows the runtime distribution with bucket size 20 and the EMD values for all three bucket sizes.

Using EMD identifies that the 6-node and 8-node configurations (Tiers 2 and 3) have the most similar query runtime distributions. If one tier should be removed, our algorithm removes the 8-node configuration (Tier 3). This choice is consistent with the PSLAs shown in Figure 2.1, where Tier 3 shows only a marginal improvement in query runtimes compared with Tier 2.

Once the PSLAManager removes that configuration, the algorithm recomputes the EMD between Tier 2 and 4. Here, for bucket sizes 5 or 10, the algorithm removes Tier 2 (6-node configuration) if a second tier should be removed because EMD(2,4) $>$ EMD(1,2). If bucket sizes are coarser-grained at 20 seconds, Tier 4 (the 16-node configuration) gets dropped instead. Using smaller-size buckets better approximates the distributions and more accurately selects the tiers to remove.

We similarly compute the EMD for the Amazon query runtime distributions using buckets of size 10. We find that EMD(Small,Medium) $= 169.10$ while EMD(Medium,Large) $= 23.67$. EMD computations with buckets of size 5 yield similar values: 168.27 and 23.75. Our algorithm removes the most expensive tier if we limit the PSLA to only two tiers. This choice is, again, consistent with the small performance gains shown for Tier 3 in Figure 2.2 compared with Tier 2.

2.3.4 Workload Clustering and Template Extraction

Once a set of k tiers has been selected, the next step of PSLA generation is workload compression: transforming the distribution of query times into a set of query templates with associated query time thresholds.

In this section, we study the behavior of the 12 different clustering techniques described in Section 2.2.5 with respect to the PSLA complexity (number of query templates) and performance error (measured as the average RMSE of the relative query times to cluster threshold times) metrics. Recall from Section 2.1 that in order to calculate the error for the PSLA as a whole, we first

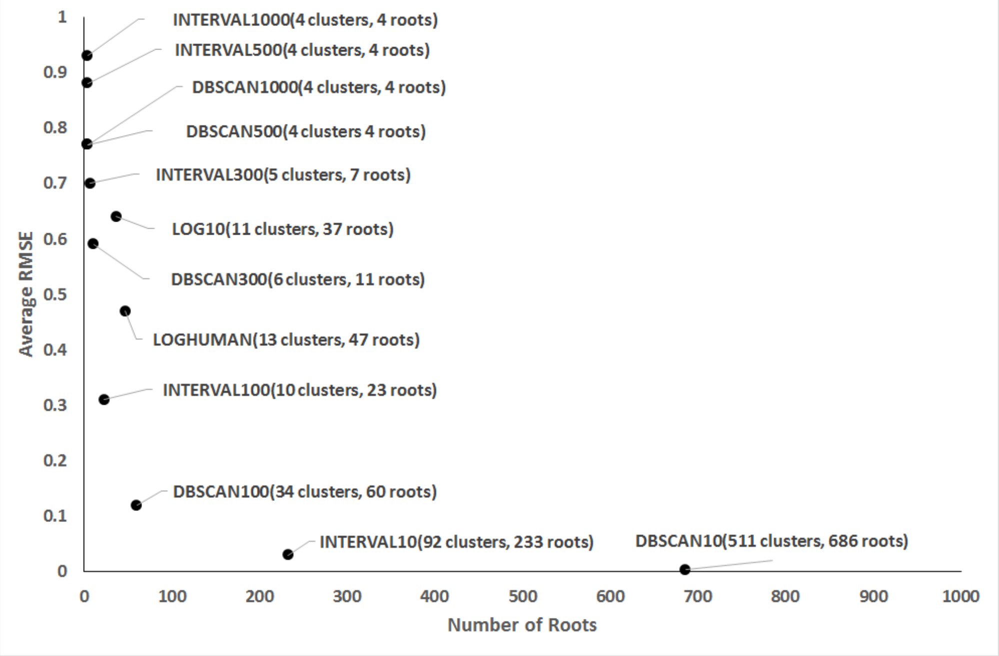


Figure 2.7: PSLA performance error (average RMSE) against complexity (# root queries) on Myria considering all four tiers.

Technique	Intra-cluster compression only			Intra-cluster and cross-tier compression		
	# of clusters	# of query roots	RMSE	# of clusters	# of query roots	RMSE
INTERVAL1000	4	4	0.93	1	1	0.90
INTERVAL500	4	4	0.88	1	1	0.82
INTERVAL300	5	7	0.70	3	5	0.33
INTERVAL100	10	23	0.31	10	21	0.10
INTERVAL10	92	233	0.03	90	221	0.01
LOG10	11	37	0.64	9	33	0.46
LOGHUMAN	13	47	0.47	9	32	0.20
DBSCAN10	511	686	0.004	448	605	0.002
DBSCAN100	34	60	0.12	24	50	0.08
DBSCAN300	6	11	0.59	5	10	0.41
DBSCAN500	4	4	0.77	2	2	0.38
DBSCAN1000	4	4	0.77	2	2	0.38

Table 2.1: Effect of workload compression on Myria. Initial workload comprises 3584 queries (896 queries in 4 tiers).

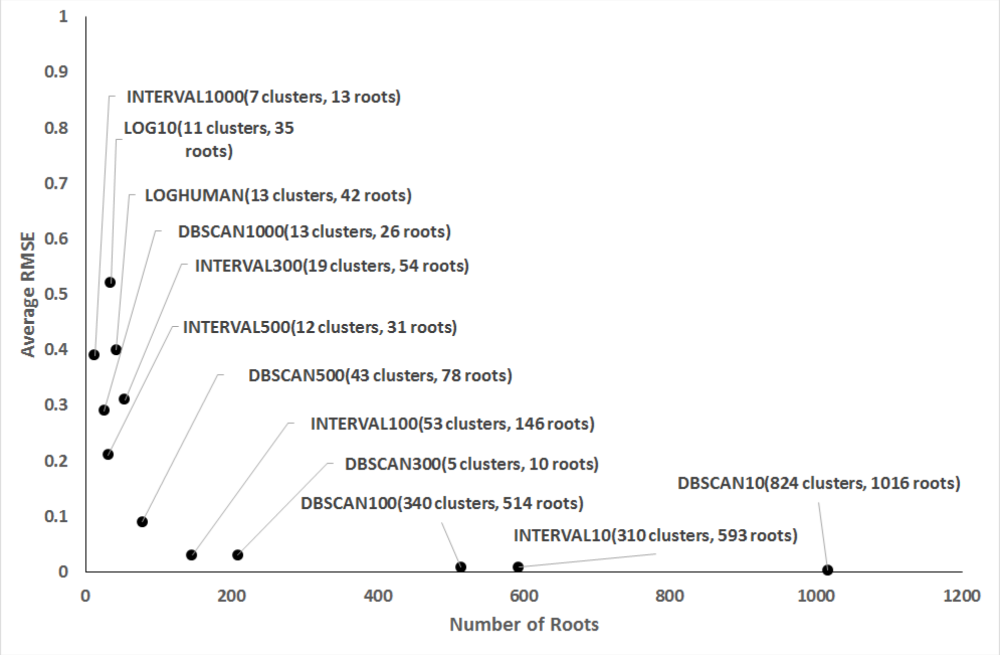


Figure 2.8: PSLA performance error (average RMSE) against complexity (# root queries) on Amazon considering all three tiers.

Technique	Intra-cluster compression only			Intra-cluster and cross-tier compression		
	# of clusters	# of query roots	RMSE	# of clusters	# of query roots	RMSE
INTERVAL1000	7	13	0.39	7	12	0.19
INTERVAL500	12	31	0.21	12	29	0.09
INTERVAL300	19	54	0.31	18	49	0.06
INTERVAL100	53	146	0.03	53	135	0.01
INTERVAL10	310	593	0.008	307	579	0.003
LOG10	11	35	0.52	10	34	0.41
LOGHUMAN	13	42	0.40	10	32	0.22
DBSCAN10	824	1016	0.003	748	941	0.001
DBSCAN100	340	514	0.008	266	429	0.003
DBSCAN300	120	209	0.03	87	142	0.02
DBSCAN500	43	78	0.09	35	64	0.06
DBSCAN1000	13	26	0.29	12	26	0.19

Table 2.2: Effect of workload compression on Amazon. Initial workload comprises of 2688 queries (896 queries in 3 tiers).

calculate the RMSE for each individual cluster. Then, we take the average of all these RMSEs across all clusters in all tiers. Again, we use real query runtimes in this section. Additionally, we disable cross-tier compression. We defer the evaluation of that step to Section 2.3.5.

Figure 2.7 and Figure 2.1 show the results for Myria for all four service tiers. As the figure shows, small fine-grained clusters (INTERVAL10 and DBSCAN10) give the lowest error on runtime (average RMSE of 0.03 or less), but yield a high complexity based on the number of clusters (92 or more) and templates (233 or more). On the other hand, coarse-grained clusters generate few clusters and templates (only 4 for both INTERVAL1000 and DBSCAN1000) but lead to higher RMSEs (0.93 for INTERVAL1000 and 0.77 for DBSCAN1000). With both methods, interval sizes must thus be tuned to yield a good trade-off. For example, INTERVAL100 yields a small number of clusters and templates (10 clusters and 23 templates) with still achieving a good RMSE value of 0.31. DBSCAN can similarly produce PSLAs with few clusters and templates and a low RMSE, as seen with DBSCAN100. In contrast, LOG10 and LOGHUMAN produce good results similar to the tuned INTERVAL100 and DBSCAN300 configurations, though with somewhat worse RMSE values, but with no tuning required.

Figure 2.8 and Table 2.2 show the results for Amazon. We observe similar trends as for Myria. In general, most of the clustering algorithms for the Amazon service have higher complexity but lower RMSE values than the same algorithms for Myria primarily because the query runtimes are more widely spread with the Amazon service. Importantly, the optimal settings for the INTERVAL and DBSCAN methods are different for this service. The best choices are INTERVAL300 or INTERVAL500 and DBSCAN500 or DBSCAN1000. In contrast, LOG10 and LOGHUMAN still yield a good trade-off between complexity (only 35 and 42 templates respectively) and performance error (0.52 and 0.40 respectively).

The key finding is thus that all these clustering techniques have potential to produce clusters with a good performance error and a low complexity. INTERVAL-based and DBSCAN-based techniques require tuning. The LOG-based methods yield somewhat worse RMSE metrics but do not require tuning, which makes them somewhat preferable. Additionally, root queries and the associated query templates effectively compress the workload in all but the finest clusters. For example, for the LOG10 clustering methods, the 896 queries are captured with only 35-37 root queries with both services, which correspond to only 4% of the original workload.

2.3.5 Benefits of Cross-tier Compression

During workload compression, once a tier is compressed into a set of clusters and templates, the PSLAManager moves on to the next tier. Before clustering the data at that tier, it drops the queries whose runtimes do not improve, , all queries that fall in the same cluster as in the cheaper tier. In this section, we examine the impact of this cross-tier compression step. Tables 2.1 and 2.2 show the result for all 12 clustering methods.

In the case of Myria, we find that cross-tier compression can reduce the number of query templates by up to 75%, as seen for INTERVAL1000. In general, most of the merging occurs from Tier 2 to Tier 1 and from Tier 3 to Tier 1, which confirms the tier selection results, where EMD values cause Tier 3 to be dropped first, followed by Tier 2.

In the three-tier Amazon configuration, we find that cross-tier compression can reduce the number of queries by up to 32%. All of the merging occurs between the medium and large tiers only and nothing is merged between the medium and small tiers. This confirms the fact that queries between large and medium tiers have similar runtimes, as computed by the EMD metric in Section 2.3.3.

As the PSLAManager drops queries whose runtime does not improve, the clusters that are created for the higher tiers also have lower RMSE values as shown in Tables 2.1. and 2.2

Cross-tier compression is thus an effective tool to reduce PSLA complexity and possibly also PSLA error.

2.3.6 Query Time Predictions

We now consider the problem of query runtime predictions. We study the implications of errors in query runtime estimates on the generated PSLAs in the next section (Section 2.3.7).

Accurate query time prediction is a known and difficult problem. This step is not a contribution of this chapter. However, we need to evaluate how errors in query time predictions affect the generated PSLA.

Based on prior work [49], we build a machine-learning model (M5Rules) that uses query features from the query optimizer to predict the runtime for each query in our generated workload. We learn the model on the synthetic database generated using the Parallel Data Generation Framework tool [108], which we call the PDGF dataset. We then test the learned model on the SSB benchmark database. Both are 10GB in size. We build separate models for Amazon and for Myria.

In the case of Amazon, we learn a model based on features extracted from the SQL Server query optimizer. We use the following features: estimated number of rows in query result, estimated total IO, estimated total CPU, average row size for query output, estimated total cost. We also add the following features as we found them to improve query time estimates: the number of tables joined in the query, the total size of the input tables, and the selectivity of the predicate that we apply to the fact table.

To build the model, we first use our workload generator to generate 1223 queries for the PDGF database. We then use the SQL Server query optimizer to extract the features for these queries. We finally execute each of these queries on each of the three Amazon configurations (small, medium, and large). We execute each query one time.

In the case of the Amazon cloud, a complicating factor for accurate query time prediction is simply the high variance in query runtimes in this service. To estimate how much query times vary across executions, we run all queries from the SSB benchmark three times on each of the three configurations in Amazon. For each query, we compute the $\frac{(\max(\text{time})-\min(\text{time}))}{\max(\text{time})}$ over the three runs. Table 2.3 shows the average result for all queries. As the table shows, query runtimes can easily vary on average by 15% (small instance) and even 48% (large instance).

This high variance in query times de-emphasizes the need for an accurate query time predictor and emphasizes the need for the cloud service to handle potentially large differences between predicted and actual query times.

We now evaluate the quality of the query time predictions. Given the large differences in query times across executions, we evaluate the model on each run separately. We also evaluate

Configuration	**Average Runtime Variation**
Small	0.149
Medium	0.181
Large	0.481

Table 2.3: Average variation in query runtimes across three executions of all SSB queries for each of the three Amazon instances. We compute the runtime variation for each query as $\frac{\max(\text{time})-\min(\text{time})}{\max(\text{time})}$ across three executions of the query.

the predictions on the median runtime across the three executions (separately computed for each query). Table 2.4 shows the results. We plot both the correlation coefficient and the relative absolute error of the model. The correlation coefficient is high for all runs, ranging from 0.864 to 0.999. The relative absolute error represents how much the predictions improve compared to simply predicting the average. In this case, the lower the percentage, the better. This error ranges from 4% to 24%, with the medium-size instance having the largest prediction errors.

These results show that, for simple queries and without indexes, a simple model can yield good query time predictions. Accurate query time predictions are difficult and, in the cloud, are further complicated by large variations in query times across executions.

We next evaluate query time predictions for the Myria service. Here, the problem is simpler because we measure the runtimes in an isolated setting, where there are no other processes competing for resources on the cluster. Additionally, we measure all query times on a cold cache.

To build the model, we execute each query from the PDGF database once. To test the model, we execute each query from the SSB database once as well. In the case of the Myria service, we partition the fact table across workers and replicate all dimensions tables. As a result, all joins become local joins that get pushed to the PostgreSQL instances, which Myria uses as per-node storage. For the feature vector, we use the same set of features as for Amazon plus the estimated query cost from the PostgreSQL optimizer. Table 2.5 shows the results. Interestingly, while the correlation coefficients are high, the relative absolute errors are no better than for Amazon. They range between 17% and 24%. These results could likely be significantly improved. For example,

		Correlation Coefficient	Relative Absolute Error
Small			
	Run 1	0.999	4.09%
	Run 2	0.999	7.19%
	Run 3	0.980	15.27%
	Median	0.955	20.13%
Medium			
	Run 1	0.864	24.28%
	Run 2	0.874	23.87%
	Run 3	0.882	24.63%
	Median	0.868	24.16%
Large			
	Trial 1	0.933	14.90%
	Trial 2	0.928	13.65%
	Trial 3	0.932	16.14%
	Median	0.953	24.14%

Table 2.4: Error on predicted runtimes for each of three runs on Amazon and for the median runtime across the runs.

our current model does not take into account possible skew in workload across workers.

2.3.7 *Effect of Query Time Prediction Inaccuracies*

Finally, we generate the full PSLAs (all tiers) for both Amazon and Myria using the predicted query times instead of the real runtimes. Figures 2.9 and 2.10 show the result.

For Myria, the PSLA using predicted times has fewer templates than the PSLA with real runtimes (Figure 2.1). A closer look at the templates from Figure 2.9 shows that the query runtimes appear to be underestimated in many cases: Several templates appear with lower time thresholds.

Since query runtimes are underestimated, more merging from the more expensive tiers to the cheaper tiers also occur, resulting in a PSLA with lower complexity. In fact, between these two PSLAs, the number of query templates drops from 32 down to 14.

We observe a similar trend for Amazon with several templates appearing with lower time thresholds and more cross-tier compression in the PSLA based on predicted runtimes. Addition-

	Correlation	Relative Absolute Error
4 Workers	0.909	24.05%
6 Workers	0.921	21.79%
8 Workers	0.951	17.24%
16 Workers	0.951	17.24%

Table 2.5: Error on predicted runtimes for the Myria service.

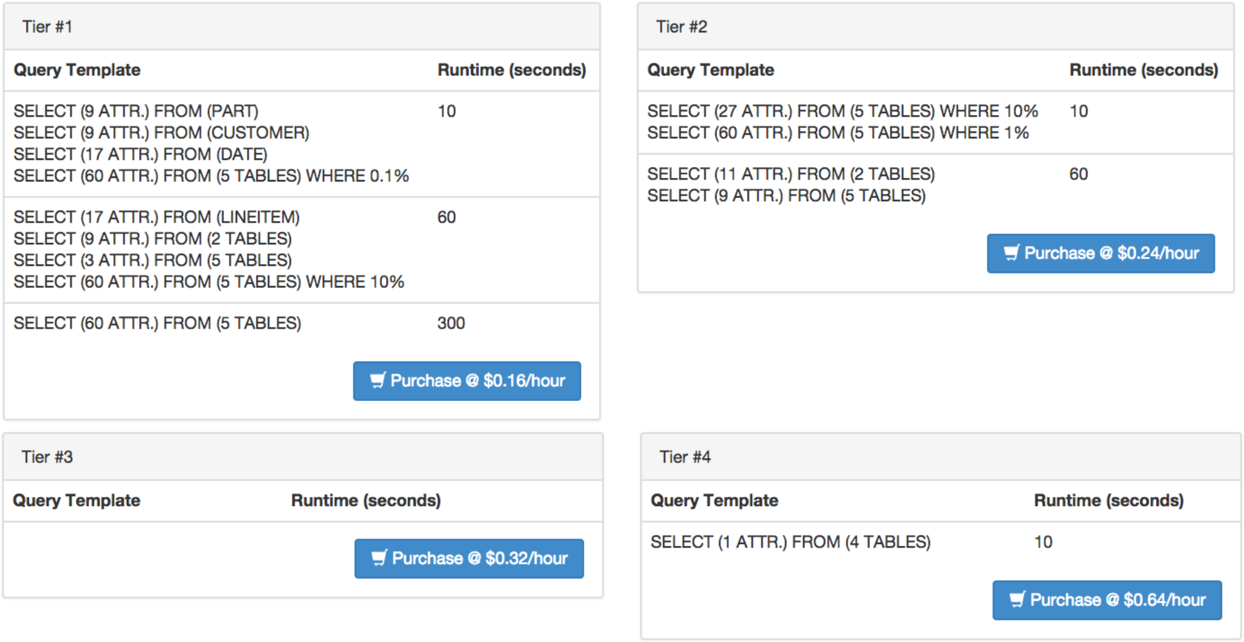
ally, for Amazon, the PSLA generally shows more grouping in the query times. Complexity decreases from 32 templates down to 11 between the PSLA that uses real runtimes and the one with predicted runtimes.

Given these results, it is worthwhile to consider the trade-offs between overestimating and underestimating the runtimes for a particular service. If runtimes are slightly overestimated, there are no surprises for the user in terms of queries not meeting their deadlines. On the other hand, this might hurt the cloud provider since the PSLA will portray runtimes that are slower than what the service can actually deliver. In contrast, underestimated query times might disappoint the user if queries end up being slower than what the PSLA indicated.

2.3.8 PSLA Generation Time

The PSLAManager can generate the concrete PSLAs shown in Figures 2.9 and 2.10 in a short amount of time. For Myria, it takes approximately 27sec to generate a PSLA for the SSB dataset. In the Amazon case, it takes an even short amount of time. Only approximately 12sec.

Table 2.6 shows the runtime for each step involved in generating the PSLA for the Myria service. The most expensive step in the PSLA generation is the cross-tier compression step, which takes approximately 19.7sec for the Myria PSLA, or approximately 73% of the total PSLA generation time. Predicting runtimes takes approximately 2sec per tier. In this example, we consider only four tiers. For some services, we could consider many more tiers. For example, all clusters of size 1 through 20, in which case query time prediction could become a significant overhead. We observe, however, that this step is easy to parallelize, which would keep the runtime at 2sec



Tier #1

Query Template	Runtime (seconds)

SELECT (9 ATTR.) FROM (PART)
SELECT (9 ATTR.) FROM (CUSTOMER)
SELECT (17 ATTR.) FROM (DATE)
SELECT (60 ATTR.) FROM (5 TABLES) WHERE 0.1%
	10

SELECT (17 ATTR.) FROM (LINEITEM)
SELECT (9 ATTR.) FROM (2 TABLES)
SELECT (3 ATTR.) FROM (5 TABLES)
SELECT (60 ATTR.) FROM (5 TABLES) WHERE 10%
	60

SELECT (60 ATTR.) FROM (5 TABLES)	300

Tier #2

Query Template	Runtime (seconds)

SELECT (27 ATTR.) FROM (5 TABLES) WHERE 10%
SELECT (60 ATTR.) FROM (5 TABLES) WHERE 1%
	10

SELECT (11 ATTR.) FROM (2 TABLES)
SELECT (9 ATTR.) FROM (5 TABLES)
	60

Tier #3

Query Template	Runtime (seconds)

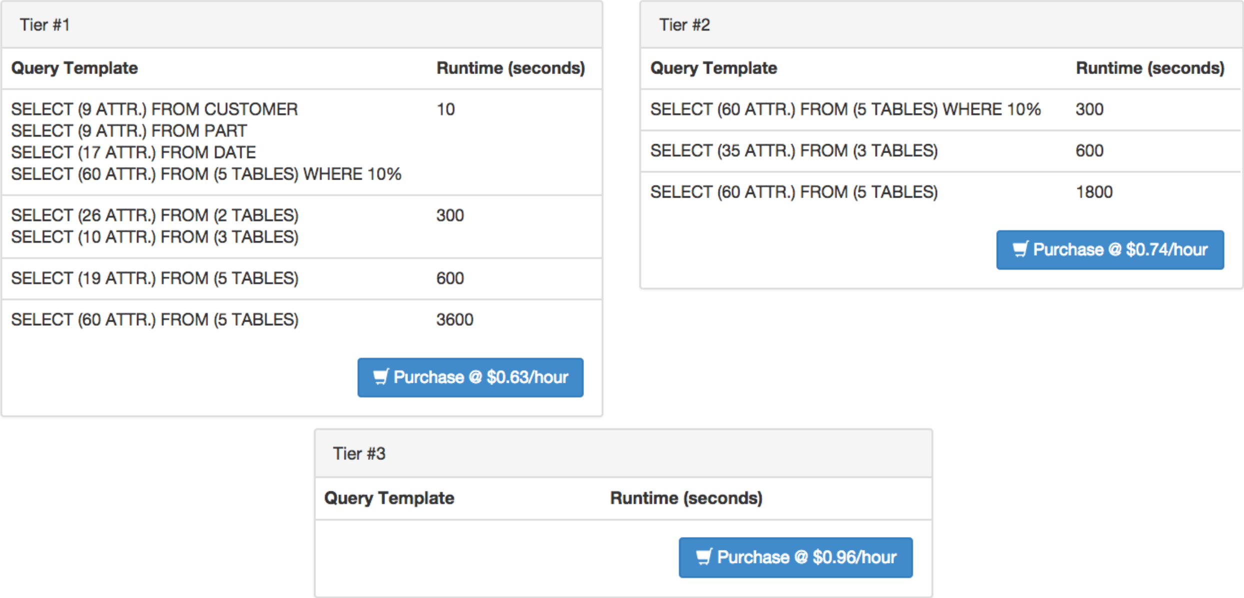
Tier #4

Query Template	Runtime (seconds)
SELECT (1 ATTR.) FROM (4 TABLES)	10

Figure 2.9: Example Personalized Service Level Agreement (PSLA) for a 10GB instance of the Star Schema Benchmark and the shared-nothing Myria DBMS service based on predicted runtimes.

if we simply process all tiers in parallel and lower for an even higher degree of parallelism. For tier selection, it takes less than 2msec to compute each EMD distance using bucket sizes of either 10sec or 5sec. The runtime of this step depends on the total number of buckets, which is small in this scenario. Finally, clustering and template extraction is fast when using a threshold-based method such as LOGHUMAN. It only takes 100msec for the most expensive Tier 1. Tier 1 is most expensive because subsequent tiers benefit from cross-tier compression before clustering. Query generation also takes a negligible amount of time compared to the other steps. Only 2msec in total.

We see a similar trend in the Amazon case. Table 2.7, shows the runtimes for each step in the PSLA process. Query generation takes approximately the same amount of time as in the Myria case, as expected. For predictions, it takes less than 1sec to predict the runtimes for each tier. Amazon is faster since it uses fewer features to predict the runtimes per query compared with Myria. On the other hand, it takes slightly longer to compute the EMDs for Amazon since the



Tier #1

Query Template	Runtime (seconds)

SELECT (9 ATTR.) FROM CUSTOMER
SELECT (9 ATTR.) FROM PART
SELECT (17 ATTR.) FROM DATE
SELECT (60 ATTR.) FROM (5 TABLES) WHERE 10%
	10

SELECT (26 ATTR.) FROM (2 TABLES)
SELECT (10 ATTR.) FROM (3 TABLES)
	300

SELECT (19 ATTR.) FROM (5 TABLES)	600
SELECT (60 ATTR.) FROM (5 TABLES)	3600

Tier #2

Query Template	Runtime (seconds)
SELECT (60 ATTR.) FROM (5 TABLES) WHERE 10%	300
SELECT (35 ATTR.) FROM (3 TABLES)	600
SELECT (60 ATTR.) FROM (5 TABLES)	1800

Tier #3

Query Template	Runtime (seconds)

Figure 2.10: Example Personalized Service Level Agreement (PSLA) for a 10GB instance of the Star Schema Benchmark and the single-node Amazon SQL Server instance based on predicted runtimes.

distributions of runtimes per tier are much more widely spread, leading to histograms with larger numbers of buckets. Again, the most expensive step is the cross-tier compression. Although it is faster than for the Myria PSLA due to the smaller number of tiers, this step takes approximately 7.5sec or 62% of the PSLA generation time.

2.4 Summary

In this chapter, we developed an approach to generate Personalized Service Level Agreements that shows users a selection of service tiers with different price-performance options to analyze their data using a cloud service. We consider the PSLA an important direction for making cloud DMBSs easier to use in a cost-effective manner.

		Average Runtime (Milliseconds)	Standard Deviation
Query Generation			
	1 Table	0.21	0.40
	2 Tables	0.45	0.59
	3 Tables	0.98	0.87
	4 Tables	1.34	0.58
	5 Tables	2.07	0.84
Predictions			
	Tier 1	1313.36	21.01
	Tier 1 & 2	2562.98	29.68
	Tier 1, 2 & 3	5894.43	69.10
	Tier 1, 2, 3 & 4	7501.20	74.72
EMD	38 Buckets (10 sec)	1.5	.002
	76 Buckets (5 sec)	0.4	.004
Log-Human Intra-Cluster Compression			
	Tier 1	102.51	2.68
	Tier 2	11.77	0.14
	Tier 3	0.006	.0003
	Tier 4	1.73	0.02
Log-Human Cross-Tier Compression			
	Tier 2 to Tier 1	9665.69	69.92
	Tier 3 to Tier 1	6733.89	99.00
	Tier 4 to Tier 1	3386.46	62.57
	Tier 3 to Tier 2	0.87	0.07
	Tier 4 to Tier 2	0.91	0.14
	Tier 4 to Tier 3	0.46	0.23

Table 2.6: PSLAManager runtime broken into its main components. The runtimes shown are for the Myria PSLA.

		Runtime (Milliseconds)	Standard Deviation
Query Generation			
	1 Table	0.23	0.44
	2 Tables	0.53	0.53
	3 Tables	0.71	0.59
	4 Tables	1.37	1.10
	5 Tables	1.97	0.94
Predictions			
	Tier 1	859.86	13.720
	Tier 1 & 2	1834.76	20.032
	Tier 1, 2 & 3	2990.18	34.288
EMD	315 Buckets (10 sec)	68.1	7.3
	629 Buckets (5 sec)	527.5	23.2
Log-Human Intra-Cluster Compression			
	Tier 1	151.60	2.98
	Tier 2	61.78	0.83
	Tier 3	0.004	0.007
Log-Human Cross-Tier Compression			
	Tier 2 to Tier 1	3978.60	43.87
	Tier 3 to Tier 1	3297.08	70.37
	Tier 3 to Tier 2	277.76	4.52

Table 2.7: PSLAManager runtime broken into its main components. The runtimes shown are for the Amazon PSLA .

Chapter 3

SLAORCHESTRATOR: REDUCING THE COST OF PERFORMANCE SLAS FOR CLOUD DATA ANALYTICS

The performance-based SLAs provided by PSLAManager are convenient for users, but they risk increasing the service costs for the cloud provider. The increase in cost comes from either SLA violations, or over-provisioning of resources to avoid these SLA violations. In this chapter, we seek to develop techniques to lower those costs. This work originally appeared at USENIX ATC'18 [98].

As we introduced in Chapter 2, users are provided with a performance-centric SLA based on their data. From the cloud's perspective, these SLAs potentially increase operational costs to the provider due to SLA violations and over-provisioning. In this chapter, we address the problem of reducing these costs. We develop SLAOrchestrator, a system that enables a cloud provider to offer query-level, performance SLAs for ad-hoc data analytics. Instead of relying on outside-generated SLAs [33, 32, 139], SLAOrchestrator uses our PSLAManager from Chapter 2 to show the user what is possible and the price tag associated with various options. In this chapter, we introduce a new sub-system called PerfEnforce that schedules queries and provisions resources all in a way that successfully brings down the cost, close to (or even below) that of the original service without SLAs. Overall, SLAOrchestrator dramatically reduces service costs for the cloud given per-query performance-based guarantees.

Figure 3.1 shows our system in action given a set of random tenants and EC2 prices.[1] The x-axis shows time and the y-axis shows the ratio of the service cost with SLAs to the service cost without SLAs. When we add performance SLAs to Amazon EMR and let the cloud provision the number of Virtual Machines (VMs) purchased under the covers, costs grow dramatically either

[1] We present the detailed experimental setup in Section 3.4 and the exact SLA function in subsection 3.2.1.

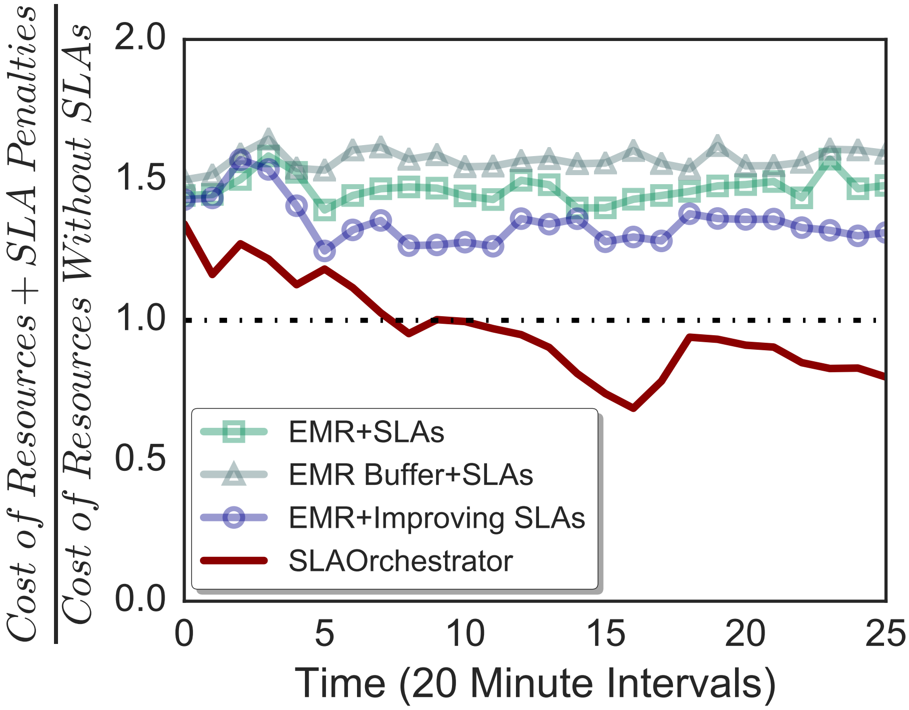


Figure 3.1: A time-changing set of tenants executes ad-hoc, analytical queries subject to performance SLAs. Static resource allocation (EMR+SLAs), even with a buffer (EMR+SLA+Buffer) leads to large cost increases. Our improving SLAs (EMR+Improving SLAs), especially with multi-tenancy and our other optimizations (SLAOrchestrator), bring costs down dramatically.

due to SLA violations (EMR+SLAs) or over-provisioning (EMR+SLAs+Buffer). Since guarantees depend on the quality of the SLAs (measured by how close runtime estimates are to the real runtimes on the purchased resources), a key component of our approach is *to improve SLAs over time* (EMR+Improving SLAs). We complement these improving SLAs with *novel resource scheduling and provisioning algorithms* that minimize costs due to over- or under-provisioning given a per-query SLA (SLAOrchestrator).

SLAOrchestrator minimizes the cost of resources and violations through three key techniques that form the core contributions of this work.

First, SLAOrchestrator is designed on the core idea of a double nested, learning loop. In the outer loop, every time a tenant arrives, the system generates a performance SLA given its current model of query execution times. That model improves over time as more tenants use the system. The SLA is in effect for the duration of a *query session*, which is the time from the moment a user purchases an SLA and issues their first query until the user stops their data analysis and leaves the system. In the inner loop, SLAOrchestrator continuously learns from user workloads to improve query scheduling and resource provisioning decisions and reduce costs during query sessions. To drive this inner loop, we introduce a new subsystem, that we call *PerfEnforce*. We present the overall system architecture in Section 3.1.

Second, the PerfEnforce subsystem comprises a new type of query scheduler. Unlike traditional schedulers, which must arbitrate resource access and manage contention, PerfEnforce's scheduler operates in the context of seemingly unbounded, elastic cloud resources. Its goal is *cost-effectiveness*. It schedules queries in a manner that minimizes over- and under-provisioning overheads. We develop and evaluate four variants of the scheduler. The first variant is based on a PI controller. Two variants model the problem as either a contextual or non-contextual multi-armed bandit (MAB) [123]. The last variant models the problem as an online learning problem. We present the query scheduler in Section 3.2.

Third, PerfEnforce also includes a new resource provisioning component. We evaluate two variants of resource provisioning: The first one strives to maintain a desired resource utilization level. The other one observes tenant query patterns and adjusts, accordingly, both the size of the overall resource pool and the tuning parameters of the query scheduler above. We present the resource provisioning algorithms in Section 3.3.

We evaluate all techniques in Section 3.4. As Figure 3.1 shows, SLAOrchestrator is able to eliminate any surcharge associated with performance guarantees, and even beats the original prices, of systems such as Amazon EMR that isolate tenants in different VMs. We also find that SLAOrchestrator outperforms various, simpler, multi-tenant deployments (Figure 3.10).

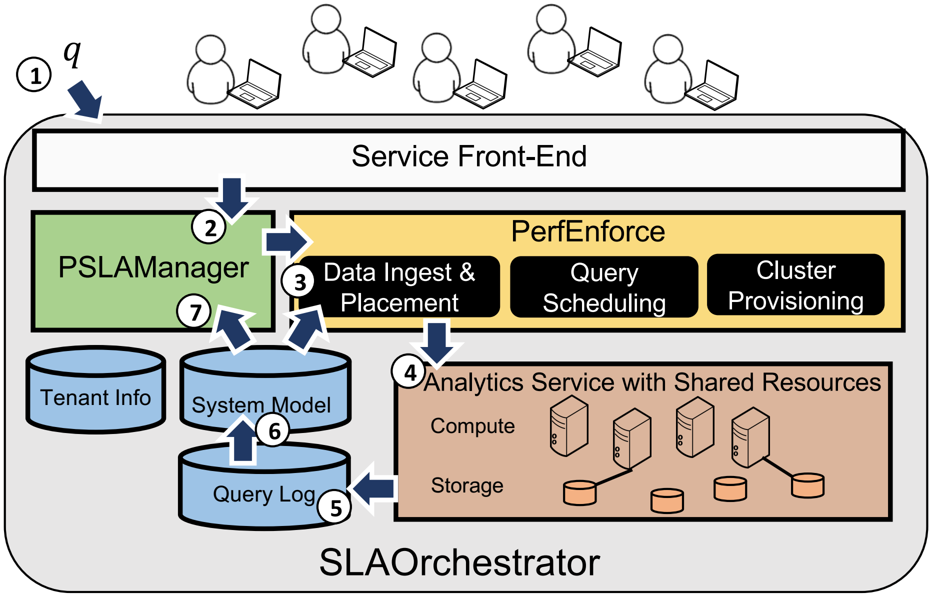


Figure 3.2: SLAOrchestrator Architecture.

3.1 System Architecture

We begin with an overview of the SLAOrchestrator architecture and a description of how all its components fit together. Figure 3.2 shows SLAOrchestrator's system architecture. In this section, we present the details of that architecture and SLAOrchestrator's double nested learning loop.

3.1.1 System Components

SLAOrchestrator runs on top of a distributed, shared-nothing, data management and analytics engine (Analytics Service) such as Spark [17] or Hive [18]. We use our own Myria system [131] in the evaluation. Similar to how tenants use Amazon EMR today, in SLAOrchestrator, tenants upload their data to the service and analyze it by issuing declarative queries. While modern systems support complex queries, in this chapter, we focus on relational select-project-join queries as proof-of-concept. However, there is nothing in our approach that precludes more complex queries in principal. On top of the Analytics Service, SLAOrchestrator includes an SLA generator (PSLA-Manager [96]), which generates performance SLAs for tenants. It also contains a dynamic scaling engine (PerfEnforce), which drives the scheduling and provisioning decisions for the underlying

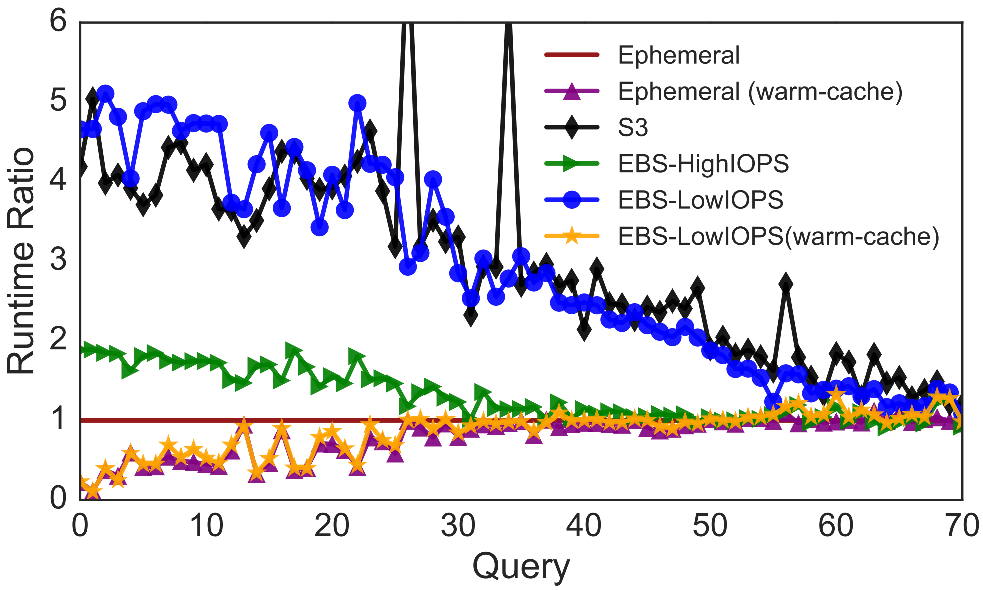


Figure 3.3: Runtimes Compared to Local Storage.

Analytics Service.

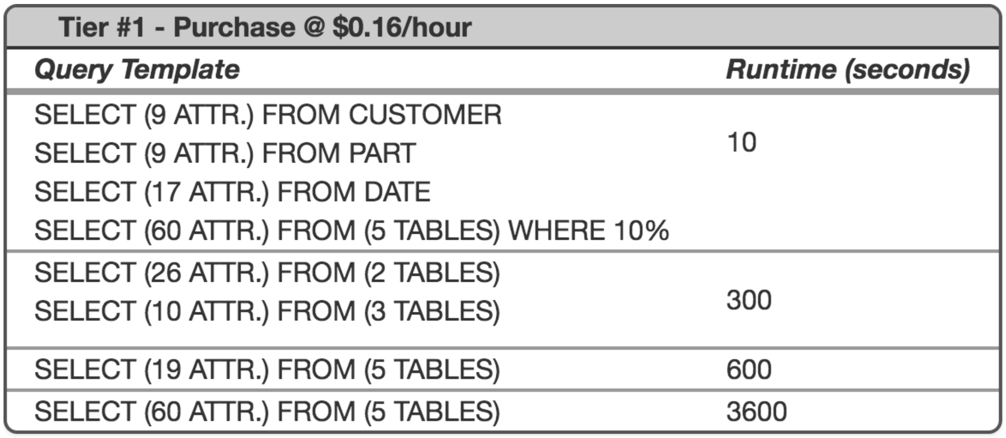
Analytics Service The back-end Analytics Service executes on a dynamically resizable pool of virtual machines (VMs) running a resource manager such as YARN [16]. PerfEnforce uses that engine in a multi-tenant fashion and takes over all query scheduling decisions. When a tenant executes a query, PerfEnforce's query scheduling algorithm determines the number of containers needed to run the query. It then allocates that number of containers from the shared VM pool. Additionally, PerfEnforce's resource provisioning determines when to grow or shrink the pool.

Analytics Service: Tenant Isolation As is common in today's big data systems, each parallel partition of each query is a task that executes in a separate container. Each query submitted by each tenant thus gets allocated its own set of containers across the VMs. Furthermore, our design partitions each tenant's dataset and attaches individual data partitions to containers, allowing for a more isolated environment. In our experiments, we use YARN containers. We schedule one container per VM and thus use the terms interchangeably.

Analytics Service: Storage Once a user purchases an SLA and before they can query their data, PerfEnforce prepares their data by ingesting it into fast networked storage, EBS volumes in our prototype. Figure 3.3 motivates our choice. The figure shows the median query execution times across three runs for a variety of storage options available on Amazon Web Services (AWS). The y-axis shows the runtime relative to local storage. Queries on the x-axis are sorted by local storage runtimes in ascending order. The 70 queries shown are based on a 100SF TPC-H SSB dataset on Myria [131] running on 32 i2.xlarge instances. As the figure shows, fast networked storage, such as EBS-HighIOPS, provides performance competitive with ephemeral storage, even on a cold cache query, without the need to dynamically migrate (or replicate) data fragments as VMs are added and removed from the shared pool. This type of storage is also affordable at less than 20% of the cost of a VM. Because we seek dynamism and must support data-intensive processing, fast networked storage is appealing.

During query execution, PerfEnforce attaches EBS volumes to different VMs and detaches them as needed. Each EBS volume holds a partition of the data, resulting in a standard shared-nothing configuration. To avoid data shuffling overheads due to scaling, PerfEnforce ingests multiple copies of each table. Each copy is partitioned across a subset of EBS volumes such that, when a query executes over a set of k containers, it uses the version of its data spread across k EBS volumes. We refer to our technical report for further details on EBS data placement and its negligible impact on performance [102].

SLA Generation To generate SLAs, we use a system from our prior work, the PSLAManager [96], but our system could work with others. PSLAManager takes as input a database schema and statistics associated with a database instance for a tenant (we use the term user and tenant interchangeably). It generates a performance-based SLA specific to a database instance as shown in Figure 3.4 for the TPC-H Star Schema Benchmark [92]. Each tier has a fixed hourly price, which maps to a pre-defined set of storage and compute resources, along with sets of grouped queries where each group contains a time threshold ("Runtime" in the figure). The time threshold represents the performance guarantee for its respective group of queries and corresponds to *query time estimates* made by the SLA generator for the corresponding resource configuration. For each resource configuration, we only consider varying the number of instances, but consistently use a standard network, and EBS-HighIOPS for storage across all configurations.



Tier #1 - Purchase @ $0.16/hour	
Query Template	***Runtime (seconds)***
SELECT (9 ATTR.) FROM CUSTOMER	10
SELECT (9 ATTR.) FROM PART	
SELECT (17 ATTR.) FROM DATE	
SELECT (60 ATTR.) FROM (5 TABLES) WHERE 10%	
SELECT (26 ATTR.) FROM (2 TABLES)	300
SELECT (10 ATTR.) FROM (3 TABLES)	
SELECT (19 ATTR.) FROM (5 TABLES)	600
SELECT (60 ATTR.) FROM (5 TABLES)	3600

Figure 3.4: Example performance SLA provided by PSLAManager with one service tier. Additional service tiers would show similar query templates but with different prices and performance thresholds.

Each tier represents a performance summary for a specific set of containers the service can use for tenant queries, which we call a *configuration*. Tiers can correspond to different types and numbers of containers, but we use a single type in our experiments. We refer to all possible configurations that the system can use to execute a query as the set $configs$. For example, $config = \{2, 4, \ldots, 64\}$, represents all even numbers of containers up to a maximum of 64. The system shows tiers for a pre-defined subset of these configurations. Later, it can schedule queries using the full set of configurations. The price of each tier is at least the sum of the hourly cost of the containers and network storage.

When a tenant purchases a performance SLA, she unknowingly purchases a configuration. The system starts a *query session* for the tenant and the latter starts paying the corresponding fixed hourly price. During the session, the tenant issues queries. The queries get queued up and execute one after the other, each one running in the entire set of containers in the purchased configuration. As we present in Section 3.2, PerfEnforce changes these allocations over time based on how fast they execute compared with the initial SLA time.

3.1.2 Double Nested Learning

To drive the SLA generation, SLAOrchestrator maintains a log of all past queries executed in the system. Initially, it executes queries from a 100GB dataset generated by the Parallel Data Generation Framework(PDGF) [108]. The system runs queries on all configurations that it will sell to populate the query log. With this information, SLAOrchestrator builds a model of query execution times. Each query is represented by a feature vector. Features correspond to query plan properties including the number of tables being joined, their sizes, the query cost estimates from the query optimizer, the number of containers in the configuration, etc. SLAOrchestrator learns a function from that feature vector to a query execution time. In our work, we use a simple linear model as in prior work [96, 129]. More complex models are possible but we find a simple linear model to yield good results for the select-project-join queries that we focus on in this chapter. With this model, predictions are made by learning the coefficients (a weight vector, w) [22] given the query features, x_q: $y(x_q, w) = \sum_{d=1}^{D} w_d \cdot x_{qd}$.

With our previous PSLAManager work [96], we observed that when a new tenant joins the system, estimates for that tenant's queries are likely to be inaccurate because the system has limited information about the tenant data and queries (only statistics on base data). However, as the tenant starts to execute queries, the system can quickly learn the properties of the data and can specialize its model to that data. PerfEnforce uses this information to dynamically adjust query scheduling and resource provisioning decisions in the context of an existing SLA. We call this the **Inner Learning Loop**. The effect of this learning is also that the system updates the SLA that it offers after each query session. This is SLAOrchestrator's **Outer Learning Loop**. The benefit of more precise SLAs to tenants is the overall reduction in the service cost. We use TensorFlow [4] to build this model and train on the PDGF dataset. We generate 4000 queries (500 per configuration) and record the features as well as runtimes into the System Model.

Figure 3.2 shows in more detail how SLAOrchestrator components interact with one another. Steps 1 through 6 denote the **Inner Learning Loop**: (1) Each tenant query, q is issued through the service front-end. (2) PSLAManager determines q's promised SLA time based on the service tier that the user previously purchased. (3) PerfEnforce uses query scheduling algorithms in conjunction with the System Model to determine the number of containers to schedule for q. (4) PerfEnforce schedules q on the Analytics Service. (5) The Analytics Service sends metadata about the query to the Query Log. (6) The System Model parses the Query Log metadata and stores features for the learning models. Once a tenant completes their session, SLAOrchestrator initiates the **Outer Learning Loop**. In Step 7, the PSLAManager system takes the information from the System Model and generates an improved SLA.

In the next two sections, we focus on the PerfEnforce subsystem and its query scheduling (Section 3.2) and resource provisioning (Section 3.3) algorithms, which are part of SLAOrchestrator's inner learning loop.

3.2 Dynamic Query Scheduling

Every time a new tenant purchases a service tier, PerfEnforce begins a query session for that tenant. The initial state of the query session indicates the configuration (i.e., number of containers) that corresponds to the purchased service tier. Many sessions are active at the same time and PerfEnforce receives streams of queries from these active tenants. Each query is associated with a possibly imperfect SLA. That is, the query may run significantly faster or slower than the SLA time if scheduled on the purchased set of containers. PerfEnforce's goal is to determine how many containers to actually use for each query with the goal to minimize operation costs. In this section, we present PerfEnforce's query scheduling algorithm.

3.2.1 Optimization Function

Consider a cloud service operation interval $T = [t_{start}, t_{end}]$. The total operating cost to the cloud during that interval is the cost of the resources used for the service and the cost associated with SLA violations for tenants active during that interval. Thus, PerfEnforce's goal is to minimize the following cost function :

$$\text{cost}(T) = \text{cost}_R(T) + \sum_{u \in U(T)} (\text{penalty}(u)) \tag{3.1}$$

where $U(T)$ is the set of all tenants active during time interval, T, and $cost_R(T)$, is given by:

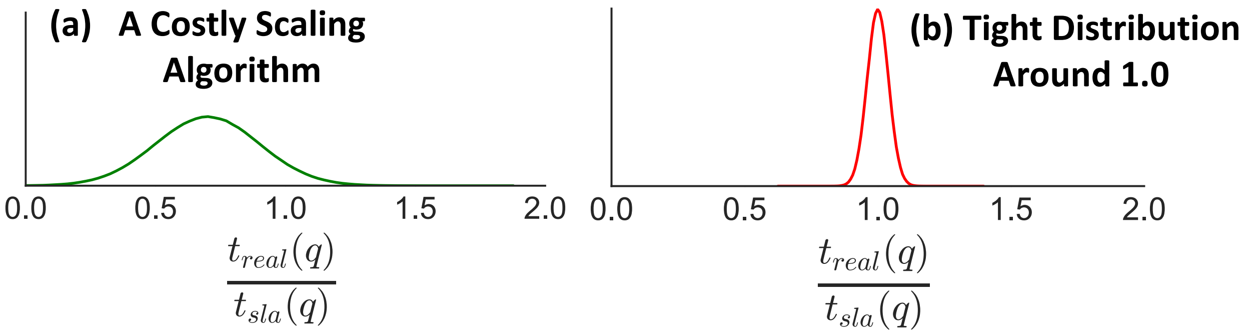


Figure 3.5: Examples of Distributions of Query Performance Ratios

$$\text{cost}_R(T) = \sum_{t=t_{start}}^{t_{end}-1} \text{cost}_t(\text{resources}) \tag{3.2}$$

where $cost_t(resources)$ represents the cost of resources for time interval $[t, t+1]$, which depends on the size and the price of individual compute instances.

The SLA penalty, $penalty(u)$, is the amount of money to refund to user u in case there are any SLA violations. In this chapter, we use the following formulation:

$$\mathcal{S}(\frac{1}{|\mathcal{W}_u|} \sum_{q \in \mathcal{W}_u} \max(0, \frac{t_{real}(q) - t_{sla}(q)}{t_{sla}(q)})) * \alpha * p_u \tag{3.3}$$

where $\mathcal{W}_u$ is the sequence of queries executed by user u, $t_{real}(q)$ is the real query execution time of query q, $t_{sla}(q)$ is the SLA time of q, p_u is the session price paid by user u in the absence of SLA violations, and α is a configurable parameter that we vary in our experiments to adjust the cost of SLA penalties compared with container resource costs. $\mathcal{S}$ is a step function that rounds up and truncates values. This step function is inspired by real SLAs in cloud services that incur penalties based on availability outages [134, 120, 124].

3.2.2 Query Scheduling Algorithms

For each query $q \in \mathcal{W}_u$ and for each user u, PerfEnforce's query scheduling algorithm must determine the number of containers from the shared pool to allocate to the query. PerfEnforce begins with using the number of containers that corresponds to the purchased service tier. It observes the resulting query runtimes and dynamically adjusts the number of containers for subsequent queries by using a *scaling* algorithm. It runs a scaling algorithm separately for each tenant.

To minimize resource costs, the scaling algorithm should schedule queries on the smallest possible number of containers. To avoid SLA penalties, however, it must schedule queries on sufficiently large numbers of containers to ensure that the real query execution time, $t_{real}(q)$, is below the SLA time, $t_{sla}(q)$. We define the *query performance ratio* as $\frac{t_{real}(q)}{t_{sla}(q)}$ and the goal of the query scheduling algorithm is thus to execute each query in the configuration that yields a performance ratio of 1.0. In practice, if the query scheduling algorithm aims for query performance ratios of X, it will yield a query performance ratio distribution around X as illustrated in Figure 3.5. To illustrate our point, we plot synthetic Gaussians. Real distributions are noisier. Since we can adjust the mean of the distribution (a.k.a. setpoint), X, the quality of the scheduling algorithm is determined by the tightness of the distribution around X. In other words, if the distribution is wide (large standard deviation σ), then the system is either wasting resources for many queries (Figure 3.5a) or causing a large number of SLA violations. A good query scheduling algorithm should yield a tight distribution as in Figure 3.5b).

Reactive Scaling Algorithms

A reactive algorithm observes errors between the real and SLA runtimes and adjusts the number of containers accordingly for each subsequent query. We implement a Proportional Integral (PI) controller and a Multi-Armed-Bandit (MAB) as our reactive methods. Both of these techniques have successfully been used in other resource allocation contexts [70, 80, 83, 86].

A limitation of the these techniques is that the configuration size chosen for a new query depends only on the rewards or errors of previous queries ignoring the features of the current query. We use the reactive methods as baseline.

Proportional Integral Control (PI). Feedback control [63] in general, and PI controllers in particular, are commonly used to regulate a system in order to ensure that it operates at a given reference point. With a PI controller, at each time step, t, the controller produces an actuator value $u(t)$. In our scenario, this is the number of containers to use for the current query. The actuator value, causes the system to produce an output $y(t+1)$ at the next time step. We compute $y(t)$ as the average query performance ratio over some time window of queries w: $y(t) = \frac{1}{|w|} \sum_{q \in w} \frac{t_{real}(q_j)}{t_{sla}(q_j)}$ where $|w|$ is the number of queries in w. The goal is for the output, $y(t)$, to be equal to some desired reference output $r(t)$, 1.0 in our setting.

The error $e(t) = y(t) - r(t)$ captures a percent error between the current and desired average runtime ratios. Since the number of containers to spin up and remove given such a percent error depends on the configuration size, we add that size to the error computation as follows: $e(t) = (y(t) - r(t))u(t)$.

The PI controller, chooses the next number of containers as a combination of the initial configuration size $u(0)$, the most recently observed error, $e(t)$, and the sum of all accumulated errors $\sum_{x=0}^{t} e(x)$. k_p and k_i are tunable controller parameters, which determine how strongly the controller reacts to recent errors and how much it weighs history: $u(t+1) = u(0) + \sum_{x=0}^{t} k_i e(x) + k_p e(t)$

Multi-Armed Bandits (MAB). In a MAB problem, the system must repeatedly choose among k different options, or *arms*. At each timestep t, the system makes a decision by selecting one of k arms, a_t, and receives a reward, r_t [123]. In our setting, each arm is a configuration from the set $configs$. The arm choice is the decision to schedule the next query using a given configuration size.

The goal is to maximize the total reward across many timesteps. In the bandit setting, the algorithm must learn the reward distributions for different arms through a process of trial and error [22]. At each timestep, the system must thus choose to either select the arm with the highest estimated reward (*exploitation*) or try another arm (*exploration*) in order to acquire more information and maximizing the reward across all timesteps [123].

To help balance between exploration and exploitation, we use a heuristic known as *Thompson Sampling* [28]. During initialization, we define priors describing the expected reward of each arm. In our setting, we do not make assumptions for each configuration. Instead, we initialize the model for each arm using a uniform distribution, a noninformative prior. At timestep t, the system constructs a posterior distribution for each arm based on observed rewards, $P(\theta | a, r_0, ..., r_{t-1})$, where

θ represents the model parameters. For each query submitted, the system samples from a candidate posterior distribution, defined as $\hat{\theta}$. Given that our prior is based on a uniform distribution, we use a t-distribution to represent our posterior. This t-distribution takes the reward mean, variance, and count as input. As the system samples from this posterior, we select the arm with the highest expected reward, $arg\,max_a E[P(r_t|\hat{\theta}_\alpha)]$.

Proactive Scaling Algorithms

To address the limitations of the reactive techniques, we consider two other scaling algorithms that both include additional context, x_q, for each incoming query, where x_q is a $D-$dimensional vector of features describing the query, $x_q = (x_{q_1}, ..., x_{q_D})^T$. To generate the feature vector, we use the query optimizer of the back-end query execution engine and include information from the query plans (e.g. number of columns, estimated costs, estimated rows, estimated width, and the number of workers scheduled to run the query).

Contextual Multi-Arm Bandit (CMAB). This approach is a variant of the multi-armed bandit problem that includes contextual information. In a CMAB problem, at each timestep t, the algorithm receives a feature vector, x_q, as input, and uses it to determine the best arm, a_t. CMAB does this by building a model *for each configuration* that predicts the reward in that configuration given a query feature vector. The expected value of the reward for each arm and feature vector thus becomes: $q_\star(a) = E[r_t|a_t, x_q, \theta]$.

Where θ represents the parameters of the generated model [28]. As with MAB, PerfEnforce uses the Thomson sampling heuristic to balance exploration and exploitation. At each timestep t, PerfEnforce builds a predictive model for each state by computing a bootstrap sample over all previous observations. PerfEnforce selects the action that corresponds to the state with the best predicted reward (i.e., reward closest to 1.0). In our prototype implementation, we use the REPTree model from Weka [56] as used in BanditDB [86]. For the first N queries in a tenant's session, we begin with a "warm-up" phase where we execute queries a small number of times in each configuration to initialize the observations for that configuration. PerfEnforce runs the "warm-up" session at the start of the query session, which could impact performance for some queries.

Online Learning The CMAB technique described above presents two practical challenges. First, it is difficult to determine the number of queries that should be used to initialize the distributions for each state. At least one query must be executed in each state, which can be either unnecessarily expensive or undesirably slow. The overhead especially penalizes short query sessions as early queries undergo larger amounts of exploration. Second, the observations collected are independent for each state. If one configuration suddenly results in slower or faster runtimes, this knowledge does not propagate to other states.

Because of the above limitations, we propose a different algorithm for our setting. We build a single model of query execution times with the configuration size as a feature. As a user executes queries, we always schedule those queries in configuration sizes expected to yield the best performance ratio and use the resulting query execution times to update our global model.

As described in the previous section, SLAOrchestrator maintains a model of query execution time that it uses for SLA generation. The idea here is for PerfEnforce to *continuously update* that model, during a tenant's query session, based on the measured query execution times. To update the model, PerfEnforce uses stochastic gradient descent. For each data point, it slowly updates the weight vector based on the gradient of a loss function, E: $w^{(\tau+1)} = w^{(\tau)} - \eta \nabla E$ [22]. Where τ represents the nth data point and η represents the learning rate. Importantly, PerfEnforce maintains a separate model of query execution time for each dataset so as to specialize its model to the properties of that dataset. If the underlying data significantly changes, the model could take time to adjust to changes, depending on the learning rate. Since we primarily focus on analytic sessions, we do not evaluate how this model adapts to updates. Training this model is relatively cheap, taking approximately 2.38s for a single epoch. Each prediction takes $\sim$10ms.

Setpoint Adjustment With all algorithms above, PerfEnforce strives to schedule queries such that their performance ratios form a tight distribution around a desired setpoint. An important question is how to tune the value of that setpoint. If the setpoint is 1.0 and the mean of the distribution falls on that setpoint, 50% of all queries will miss their SLA times. The setpoint can be lowered such that more, perhaps 90% of all queries, meet their SLA time. Lowering the setpoint, however, will increase the number of containers used for those queries and will thus raise resource costs. In SLAOrchestrator, we adjust the setpoint dynamically. We do so at the same time as we make cluster provisioning decisions as described next.

3.3 Dynamic Provisioning

With the above query scheduler, the total number of containers needed to service the active set of tenants varies over time. To reduce operation costs, PerfEnforce includes a resource provisioning component that determines when to grow and shrink the *shared* pool of compute resources. Provisioning is particularly challenging as it can take time to spin up new virtual machines. We observe that it takes approximately 12 seconds to spin up a virtual machine with a pre-loaded image on Amazon. We consider this start up time throughout our evaluation.

Utilization Provisioning: The most common approach to resource provisioning is to maintain a desired resource utilization level. Typically, this means adding (or removing) resources when CPU, I/O, network and memory usage move beyond (or below) set thresholds [14, 134, 51, 118].

We posit, however, that measuring resource utilization levels directly is not the right approach for PerfEnforce because tenants are allocated resource containers. As such, some tenants might execute I/O intensive workloads while others may run CPU intensive workloads, leading to very different resource utilization levels for various containers. In general, high resource utilization does not imply a higher demand for resources [37].

Instead of aiming for a given CPU or I/O utilization goal, we aim for an average *VM utilization* target, Z. The utilization of each machine is measured by the percentage of time it is actively running queries (wall clock time). For our target Z, we aim for an average utilization across all shared VMs, $AvgUtilization$.To determine the number of machines the system should provision to meet Z, we implement a PI controller where the set point is Z. Besides wall clock time, we note that other metrics for system state could be used as well. For example, the system could target a desired percentage of idle machines or a desired tenant query queue length.

Simulation-based Provisioning: For a more proactive approach to provisioning, we propose to explicitly consider tenant recent workloads rather than only measure resource utilization. Specifically, we propose to build models of tenant workloads and estimate the smallest number of shared resources to support these modeled workloads. This approach should be more effective than simply looking at utilization, since the latter is tightly coupled with the specific set of executed queries and the query scheduler's resource allocation decisions, which are themselves constrained by the amount of shared resources. To estimate the best number of shared resources to support tenant modeled workloads, we use simulations. This approach is not new and has been recently used in the "What-If" engine from Tempo [125], where the goal is to simulate the performance of many configurations of the MapReduce Resource Manager. We aim to understand how such a provisioning algorithm in combination with a learning query scheduler can help make profitable decisions in a multi-tenant service.

In this provisioning approach, we model each tenant, u, with a tuple (Q_u, λ_u), where Q_u is a set of queries that the tenant may issue and λ_u is the tenant's average think time between consecutive queries. PerfEnforce learns both values from a recent window of each tenant's query session. Based on these models, PerfEnforce then generates random sessions for each active tenant. To generate a random session, PerfEnforce samples queries from the recent query workload and also samples the think time based on a Poisson distribution. During these simulations, we also evaluate the costs of dynamically shifting the setpoint. In general, these simulations help PerfEnforce discover whether setpoint adjustments are necessary for active tenants or whether nodes should be added or removed to further save on costs.

3.4 Evaluation

We run SLAOrchestrator and execute all queries on Amazon EC2 using i2.xlarge (4 ECU, 30 GB Memory) instance types priced at \$0.12/hr. We consider eight types of query scheduling configurations, each with a different number of compute instances: $\texttt{configs} = \{4, 8, 12, 16, 20, 24, 28, 32\}$. For multi-tenant experiments, we run simulations with up to thousands of servers and use the query times measured on EC2. For our underlying shared-nothing, database management system, we use Myria [131]. Myria uses PostgreSQL as its storage subsystem.

To generate each tenant's query sequence, $\mathcal{W}_u$, we alternate between different patterns of

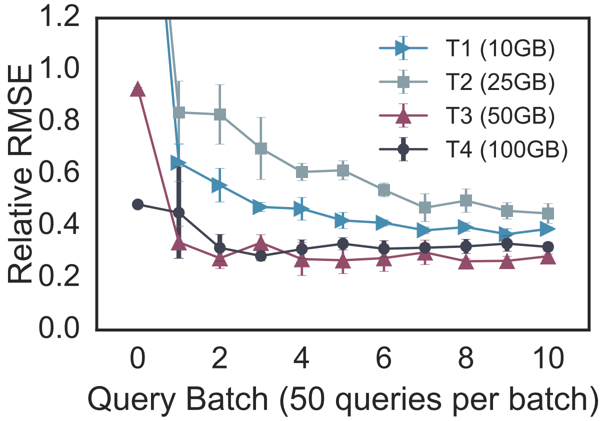


Figure 3.6: SLA improvements across query sessions

queries. For example, one tenant might run small, lightweight queries for a majority of the session before switching to queries with larger joins and higher latencies. Thus, for some random number of queries, k, we define the following three discrete distributions: (1) number of joins, (2) number of projected attributes, and (3) selectivity factor. For the next k queries, we sample from each of these distributions and generate a query that meets all the sampled characteristics. Once k queries are generated, we define new distributions for the next random interval of queries. We use both uniform and skewed (zipfian) distributions. Unless stated otherwise, all the query workloads throughout the evaluation are generated in this fashion.

3.4.1 Evaluation of SLA Predictions

A key tenet of SLAOrchestrator is the idea that the system should update SLAs because they rapidly improve as a tenant queries a database. We validate this hypothesis in this section. Figure 3.6 shows the error of the SLA predictions for four tenants, each with a different, random star schema [108] and database instance: T1 = 10GB, T2 = 25GB, T3 = 50GB, T4= 100GB. We generate a set of SPJ queries with random selection predicates for each tenant. As tenants execute queries, SLAOrchestrator updates the query time model separately for each database. After each query batch, PSLAManager re-generates an updated SLA. As the figure shows, in all cases, the RRMSEs (relative root mean squared errors) between the real runtimes and the predicted SLA

SLA	Description
Small Gaussian Error (SG SLA)	SLA assumes a good prediction model and tests sensitivity to small errors (or variance) in query times. Generated by taking the real query execution times at the purchased tier and adding a small Gaussian error: $\mathcal{N} = (\sigma = 0.1 * t_{real}(q), \mu = 0)$.
Positive/Negative Gaussian Errors for Large Joins (PLJ/NLJ SLA)	We skew SLA runtimes for some query types. We introduce large positive/negative errors to the real runtimes on queries with a large number of joins ($>$ 3 joins) and with a runtime $>$100 seconds. We update the runtime to $t_{real}(q) + \|e\|$ (or $-\|e\|$), where e is sampled from a Gaussian distribution, $\mathcal{N} = (\sigma = 0.3 * t_{real}(q), \mu = 0)$. For other queries, we still inject small errors as in SG SLA.
Initial SLA	This is the least accurate SLA, where runtimes are generated by an initial offline-trained model.

Table 3.1: SLAs used in experiments.

runtimes decreases rapidly after the first batch and then improves more slowly. We compute the error on a sample of queries generated by the PSLAManager for the tenant SLA. We measure the RRMSE as: $\sqrt{\frac{1}{|\mathcal{W}|}\sum_{q\in\mathcal{W}}(\frac{(t_{real}(q)-t_{sla}(q))}{t_{sla}(q)})^2}$. The prediction errors observed before running an initial batch of queries (Query Batch 0), are highly dependent on the similarity between the tenant's database and the synthetic database used to train our offline base model. In our experiments, while databases differ in their schemas and table sizes, we find that table sizes have the greatest impact on prediction errors. Our offline model is trained on a generated 100GB PDGF dataset. We observe a higher initial RRMSE error (approx. 2.4-2.5) for tenants T1 and T2 with the smaller databases.

3.4.2 Evaluation of Query Schedulers

The goal of each query scheduler is to ensure a tight distribution (small σ) of query performance ratios around a μ close to 1.0 (later in subsection 3.4.4, we consider dynamic setpoint tuning). In this section, we evaluate how the different scheduling algorithms perform in the face of different tenant workloads. All tenant workloads are based on the 100GB TPC-H SSB benchmark [92]. We evaluate the algorithms using different-quality SLAs as shown in Table 3.1, which could correspond, for example, to different model qualities as shown in subsection 3.4.1.

We first evaluate the PI-Control scheduling algorithm on four different SLA types and, in each case, on 10 different, randomly generated, tenant query sessions. We execute the PI controller on each tenant's query session independently and measure the resulting query performance distribu-

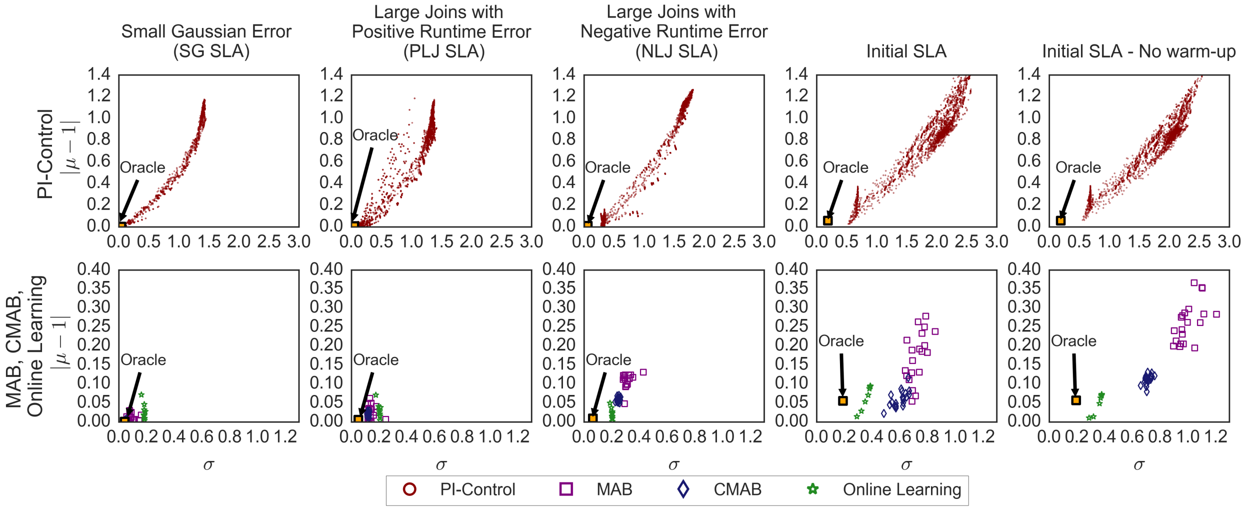


Figure 3.7: Evaluation of PI-Controller, MAB, CMAB and online learning scheduling algorithms

tion for that tenant. We then compute the average μ and σ across these 10 distributions and plot them in the first row of Figure 3.7. The y-axis represents the distance between μ and 1.0, while the x-axis displays the standard deviation of the query performance distributions. Because the PI controller has three tunable parameters (k_p, k_i and w), each point in the figure corresponds to one such parameter combination. For each graph, we also plot the average distribution of an Oracle, which always selects the best configuration size for each query. The best parameter combinations are those closest to the Oracle. If any technique's parameters result in a distribution with a higher σ or a μ farther from 1.0, this error impacts cost, which ultimately depends on the cost function. As the figure shows, for all SLAs, the PI-Control algorithm results in average distributions that are far from the average distribution of the Oracle. There are no best set of parameters that work across all workloads.

In the second row of Figure 3.7, we show the average distributions for MAB, CMAB, and online learning across the same set of SLA types and tenant query sessions. Note, the ranges for the axes are much smaller for these graphs compared to the PI-Controller, which shows that these techniques result in average distributions much closer to the Oracle. For both bandit techniques, we execute each tenant's query session 20 times due to their variance when sampling. For online learning, we vary the learning parameter, η. In the first 4 columns of the figure, we omit the performance ratios for the first 20 queries for all scaling techniques, since the bandits require an initial "warm-up" phase, where they need to try each configuration at least two times.

For the SG SLA, the bandit techniques result in average distributions nearly identical to the Oracle. Since both techniques rely on learning a distribution of query performance ratios per configuration, they quickly find the optimal configuration during the warm-up phase and select this configuration for a majority of the queries. Since the online learning technique is directly predicting the runtimes for each query, the prediction errors result in average distributions that are slightly farther away from the Oracle. For the PLJ SLA, all techniques perform similarly as most of the runtimes are meet at configurations that are close to the purchased tier. In contrast, online learning outperforms both bandit-based methods for the NLJ and Initial SLAs. The NLJ SLA underestimates runtimes, which requires the schedulers to accurately choose across a wider set of configuration options. For these more difficult cases, context is critical as the scheduling algorithm must make different decisions for different queries. There is no single best configuration. As a result, CMAB and the online learning approach both outperform the simpler MAB scheduler. Online learning further outperforms CMAB because this technique is able to quickly learn the performance correlations between configurations, which is crucial for the initial SLA as it requires scaling for each query.

The final column shows the average performance ratio distributions when using the initial SLA and including the queries in the warm-up period. As the figure shows, the online learning technique significantly outperform the bandit-based methods because it has the extra benefit of starting to learn from the offline model and learning more quickly because it learns a single model for all configurations. These results show that the PI controller is ill-suited to our problem and we do not consider it further.

We now evaluate how query scheduling algorithms can adapt to changing conditions. Recall, the goal of these query schedulers is to ensure a query performance ratio distribution close to 1.0. We generate a query sequence by selecting one query and running it repeatedly several times. Each time we run this query, we record the query performance ratio. Once we reach the 250th iteration,

	MAB	CMAB	Online Learning	Oracle
μ	1.1368	1.1244	1.0161	1.0015
σ	0.1680	0.0871	0.0522	0.0008

Table 3.2: Ratio distributions during slow down

Notation	Description
U_{init}	Initial number of tenants in the session
V_{init}	Initial number of virtual machines
$\lambda_{arrival}$	Average time between new tenants
$\lambda_{thinktime}$	Average tenant think time
$\lambda_{terminate}$	Average tenant session duration
$\mathcal{M}$	Provisioning monitoring time interval

Table 3.3: Parameters of multi-tenant experiments

we increase the query's runtime by 25% (essentially slowing down the system) for the rest of the session, running up to 1000 iterations. Table 3.2 shows that the online learning technique reacts the fastest to this change in conditions, leading to an overall mean performance ratio closest to 1.0.

3.4.3 Evaluation of Provisioning Algorithms

We first evaluate each provisioning algorithm in combination with the Oracle query scheduler to ensure that query runtime penalties are not a side-effect of the query scheduler's mispredictions. We launch each multi-tenant tenant session based on session parameters summarized in Table 3.3.

We introduce up to 100 tenants in a session and simulate a shared cluster with thousands of containers/VMs. We sample arrival times, think times, and session durations from Poisson distributions defined by their corresponding parameters $\lambda_{arrival}$, $\lambda_{thinktime}$ and $\lambda_{terminate}$. PerfEnforce always keeps at least a minimum of 4 machines launched at all times, to ensure that there are enough machines available to execute queries. Each provisioning algorithm monitors the shared resources and tenants for $\mathcal{M}$ minutes before adding or removing VMs from the pool. Our step function $\mathcal{S}$ provides no service credit if the system misses the runtime by 10%. For each additional 20% increment and given a threshold from x% to y%, we increase the credit to y%.

As described in Section 3.1, each submitted query gets allocated a set of containers. We schedule one container (running a Myria process) per VM. For each query, the system assigns the tenant's EBS volumes to a set of VMs in the pool. After the query completes, the volumes are detached from the VMs, making them available to other tenants. We find it takes 4 seconds to mount a volume to a VM. Detaching takes approximately 11 seconds. We include these delays in the experiments.

Utilization and Simulation-based Provisioning We compare the multi-tenant session costs when provisioning VMs using either the utilization-based or simulation-based approaches. We launch 100 VMs with 10 initial tenants and set the provisioning monitoring time to $20min$. Figure 3.8 shows the results and the other experimental parameters. For different average utilizations, Z, we show the results for the best parameter values V_{k_p} and V_{k_i}. The y-axis shows the cost per time unit, while the x-axis shows the value of the α parameter. Recall from Equation 3.3 that we define α as a tunable parameter that amplifies the weight of the SLA penalty compared to the resource cost. The hash pattern in each bar represents the proportion of the cost due to $Cost_R$, the cost of resources. Other costs come from SLA violations. Error bars show variance across 10 runs. As expected, the utilization-based method requires tuning depending on the α value. Simulation-based provisioning has the double-benefit of avoiding any tuning and more cost effectively provisioning shared resources compared to the utilization-based approach.

Combining Scheduling and Provisioning We now evaluate the performance of simulation-based provisioning in conjunction with various query scheduling algorithms on the initial SLA. In Figure 3.9, we vary alpha (x-axis) and measure the total cost compared with an Oracle query scheduler (y-axis). As a baseline, we also include a naive query scheduler, $static$, which simply schedules each query on the configuration initially purchased by the user. We also include utilization-based provisioning at $Z = .25$ (using online learning as the query scheduler). We still initialize the session with 10 tenants, but we start with a larger pool of 320 VMs, allowing enough room to have each initial tenant schedule queries on up to 32 containers. In this experiment, since we also include CMAB, we extend the session times to 180 to ensure the algorithm has more time to operate in steady state (beyond the warm-up phase).

Overall, simulation-based provisioning continues to outperform the utilization-based approach

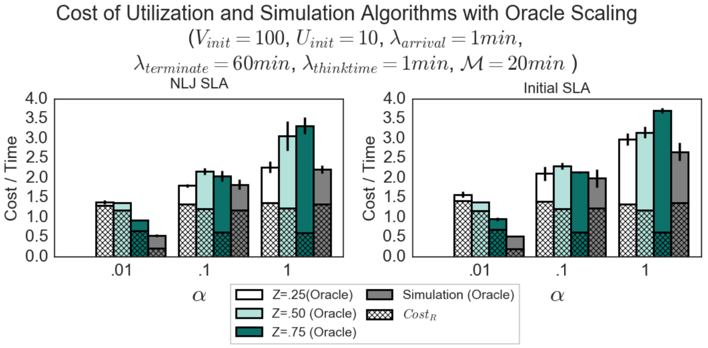


Figure 3.8: Comparing utilization-based and simulation-based provisioning in conjunction with an Oracle query scheduler. The hash pattern represents the proportion of the cost due to $Cost_R$

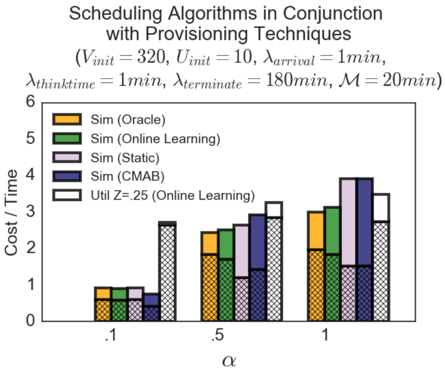


Figure 3.9: Costs of query scheduling in conjunction with provisioning

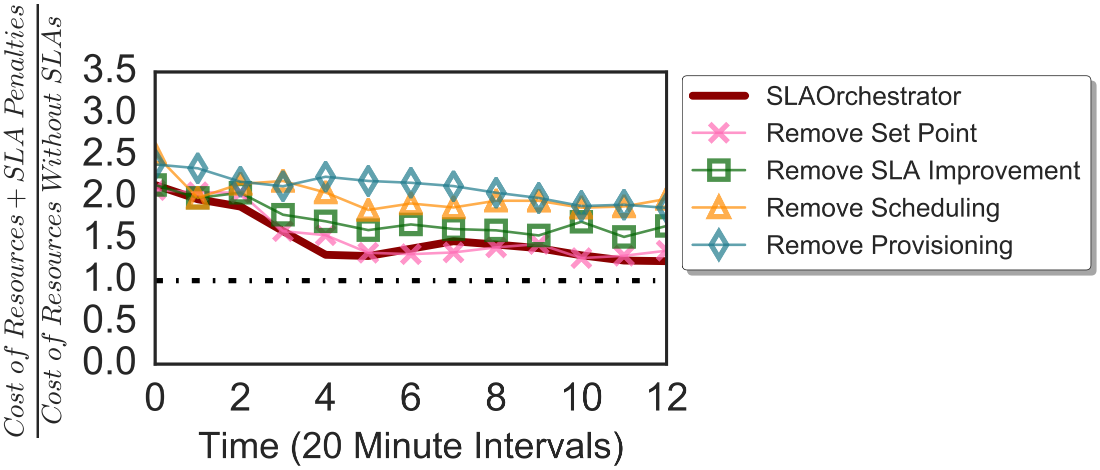
even with a less perfect scheduler. Even when penalties are high, simulation-based provisioning reduces costs by 11% and more for lower penalties. Additionally, the online learning-based scheduler yields similar costs to the Oracle scheduler (a 4% overhead). As expected, it significantly outperforms the static scheduler and CMAB when SLA penalties are expensive, with 20% cost savings. CMAB does worse because it causes more SLA violations. For small α, the CMAB approaches result in costs lower than even the Oracle scheduler. This is because the CMAB's warm-up phase initially schedules queries on all available configurations (even small configurations), which then causes the simulation approach to provision less resources. Throughout the session, resources are not added back in due to the low α value.

3.4.4 Dynamic Setpoint Tuning

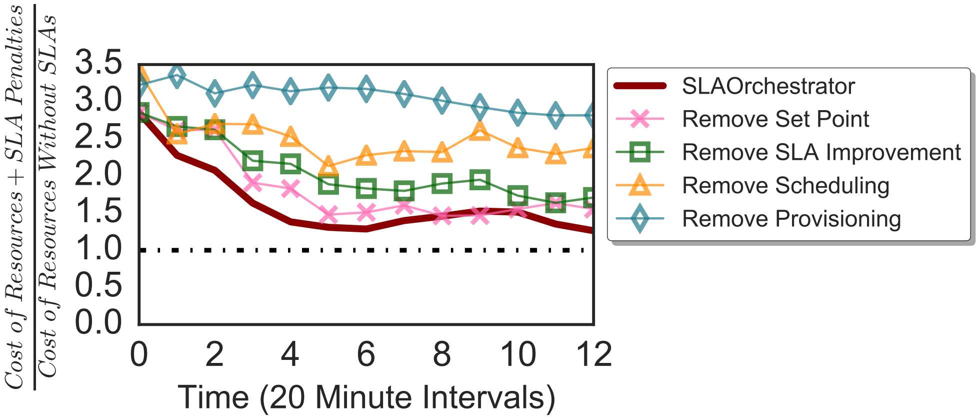
Finally, we evaluate the benefits of dynamic setpoints together with the relative benefits of the other optimizations. Figure 3.10a shows the results. In the figure, we start with SLAOrchestrator as initially shown in Figure 3.1. We then remove optimizations one at a time in order. First, we remove the ability to use dynamic setpoints, followed by removing SLA improvements, scheduling and provisioning. We remove these optimizations to run SLAOrchestrator as a simpler multi-tenant system. To emphasize the differences between optimizations, we use $\alpha = 2$ and $\alpha = 3$. In this experiment, we start the session with 5 tenants and 80 VMs (ensuring 16 nodes per tenant, the amount they have purchased). New tenants arrive approximately every 5 minutes, and tenants finish their session after 180 minutes. As seen in the figures, removing each optimization increases the cost. This is especially apparent for $\alpha = 3$, where SLAOrchestrator decreases the cost by up to 59% compared to the case with no optimizations.

3.5 Summary

We presented SLAOrchestrator, a new system designed to minimize the price of performance SLAs in cloud analytics systems. SLAOrchestrator uses a double learning loop that improves SLAs and resource management over time. To support the latter, the system also includes an efficient combination of elastic query scheduling and multi-tenant resource provisioning algorithms that work toward minimizing service costs. Experiments demonstrate that SLAOrchestrator dramatically reduces service costs for a common type of per-query latency SLAs.



(a) $\alpha = 2$



(b) $\alpha = 3$

Figure 3.10: Performance when disabling different SLAOrchestrator optimizations

Chapter 4

DEEPQUERY: AN EMPIRICAL ANALYSIS OF DEEP LEARNING FOR CARDINALITY ESTIMATION

Query optimization is at the heart of relational database management systems (DBMSs). Given a SQL query, the optimizer automatically generates an efficient execution plan for that query. Even though query optimization is an old problem [115], it remains a challenging problem today: existing database management systems (DBMSs) still choose poor execution plans for many queries [78]. Cardinality estimation is the ability to estimate the number of tuples produced by a subquery. This is a key component in the query optimization process. It is especially challenging with complex queries that contain many joins, where cardinality estimation errors propagate and amplify from the leaves to the root of the query plan. One problem is that existing DBMSs make simplifying assumptions about the data (e.g., inclusion principle, uniformity or independence assumptions) when estimating the cardinality of a subquery. When these assumptions do not hold, cardinality estimation errors occur, leading to sub-optimal plan selections [78]. To accurately estimate cardinalities, optimizers must be able to capture detailed data distributions and correlations across columns. Capturing and processing this information, however, imposes space and time overheads and adds complexity.

Recently, thanks to dropping hardware costs and growing datasets available for training, *deep learning* has successfully been applied to solving computationally intensive learning tasks in other domains. The advantage of these type of models comes from their ability to learn unique patterns and features of the data that are difficult to manually find or design [52].

Given this success, in this chapter, we ask the following fundamental question: Should we consider using deep learning for query optimization? Can a deep learning model actually learn properties about the data and learn to capture correlations that exist in the data? What is the overhead of building these models? How do these models compare to other existing machine learning techniques? In this work, we implement a variety of deep learning architectures to predict query cardinalities. Instead of relying entirely on basic statistics and formulas to estimate cardinalities, we train a model to automatically learn important properties of the data to more accurately infer these estimates. In this chapter, we seek to understand the fundamental capabilities of deep neural networks for this application domain. For this reason, we focus on the performance of basic neural network architectures.

The database community has recently begun to consider the potential of deep learning techniques to solve database problems [132]. However, there remains a limited understanding of the potential and impact of these models for query optimization. Previous work has demonstrated the potential of using deep learning as a critical tool for learning indexes [73], improving query plans [87], and learning cardinalities, specifically through deep set models [69]. However, we argue that that the accuracy should not be the only factor to consider when evaluating these models. We must also consider their overheads, robustness, and impact on query plan selection. There is a need for the systematic analysis of the benefits and limitations of various fundamental architectures.

In this experimental study, we focus on the trade-offs between the size of the model (measured by the number of trainable parameters), the time it takes to train the model, and the accuracy of the predictions. We study these trade-offs for several datasets. Our goal is to understand the overheads of these models compared to PostgreSQL's optimizer. To do this, we build several simple neural network as well as recurrent neural network models and vary the complexity by modifying the network widths and depths. We train each model separately and compare the overheads of these models to PostgreSQL and random forest models (based on an off-the-shelf machine learning model).

To summarize, we contribute the following:

- We show how deep neural networks and recurrent neural networks can be applied to the problem of cardinality estimation and describe this process in Section 4.2.

- We comparatively evaluate neural networks, recurrent neural networks, random forests, and PostgreSQL's optimizer on three real-world datasets in Section 4.3. For a known query workload, we find that, compared to PostgreSQL, simple deep learning models that are similar in space improve cardinality predictions by reducing the error by up to 98%. These models, however, come with high training overheads. We also find that, although random forest models usually require a larger amount of space, they are fast to train and are more accurate than the deep learning models.

- In Section 4.3.3, we study these models in more detail by evaluating the robustness of these models with respect to query workload changes. We find that random forest models are more sensitive to these changes compared to the neural network and recurrent neural network models. Although, decision tree models are known to have high variance and random forest models should reduce this problem [59], we still find that the neural networks are generally more robust to changes in the data.

- In Section 4.3.4, we visualize the embeddings from the models to understand what they are learning.

- Finally, we study these models from a practical perspective in Section 4.4, where we evaluate how predictions from these models improve query plan selection. We find that these models can help the optimizer select query plans that lead from 7% to 49% faster query executions. In addition, we study how *active learning* can help reduce the training overheads.

4.1 Background and Problem Statement

Many optimizers today use histograms to estimate cardinalities. These structures can efficiently summarize the frequency distribution of one or more attributes. For single dimensions, histograms split the data using equal-sized buckets (equi-width) or buckets with equal frequencies (equi-depth). To minimize errors, statistics about each bucket are also stored including but not limited to the number of items, average value, and mode [34].

These histograms are especially relevant in cases where there are simple single query predicates. For more complex predicates, the system extracts information from these histograms in conjunction with “magic constants” to make predictions [78]. Optimizers typically do not build or use multidimensional histograms or sampling due to the increased overheads [42, 138]. As the estimates from these optimizers are not theoretically grounded, propagating these estimates through each intermediate result of a query plan can result in high cardinality errors, leading to sub-optimal query plans.

In this chapter, we use PostgreSQL’s optimizer as representative of this class because it is a mature optimizer available in a popular open source system.

Our goal in this work is to apply deep learning to the cardinality estimation problem and compare the performance of this approach empirically to that of a traditional query optimizer.

We consider the following scenario: A database system is deployed at a customer’s site. The customer has a database D and a query workload Q. Both are known. We compare the following approaches:

- **Traditional:** In the pre-processing phase, we build histograms on select attributes in D. We select those attributes following standard best practices given the workload Q. Simple best practices include collecting statistics for all frequently joined columns and for non-indexed columns frequently referenced as a selection predicate, particularly if the column contains very skewed data[3]. We then measure the accuracy of cardinality estimates on queries in Q (and queries similar to Q) and the overhead of creating and storing the histograms. We measure both the time it takes to build the histograms and the space that the histograms take.

- **Deep Neural Networks:** In the pre-processing phase, we execute all queries in Q to compute their exact cardinalities. All queries in Q are instantiated with a random number of joins and selection predicates. We use the results to train deep neural networks. We encode all queries in Q into inputs for the models, and evaluate how accurately these models are able to learn the function between the input, X and the cardinality value, Y. As above, we measure the overhead of building and storing the model and the accuracy of cardinality estimates for queries in Q and queries not in Q but similar to those in Q. To compare different architectures, we build several models by varying the width and depth.

As a simplifying assumption, in this chapter, we focus on select-project-join queries and only use single-sided range predicates as selection predicates. The join predicates consist of primary key and foreign key relationships between tables, as defined by their schema. The queries are of the following form:

```
SELECT *
FROM $r_1$, $r_2$, ... $r_n$
WHERE $join_R$ and $a_1$ <= $c_1$ and $a_2$ <= $c_2$ and ... $a_n$ <= $c_n$
```

where the FROM clause defines a set of relations from dataset D and $join_R$ represents the corresponding join predicates. The WHERE clause describes a set of single-sided range selection predicates, where a_i represents an attribute and c_i represents a constant value.

4.2 Machine Learning-Based Cardinality Estimation

The first contribution of this chapter is to articulate how to map the cardinality estimation problem into a learning problem. We show the mapping for three types of models: neural networks, recurrent neural networks, and random forests.

For ease of illustration, in this section, we use a simple running example comprising a database D with three relations, $D : \{A, B, C\}$. Each relation has two attributes where relation A contains $\{a_1, a_2\}$, relation B has attributes $\{b_1, b_2\}$, and relation C has attributes $\{c_1, c_2\}$. In this database, there are two possible join predicates. Attribute a_2 is a foreign key to primary key attribute b_1, while b_2 serves as a foreign key to primary key attribute c_1.

4.2.1 Neural Networks

Deep learning models are able to approximate a non-linear function, f [52]. These models define a mapping from an input X to an output Y, through a set of learned parameters across several layers

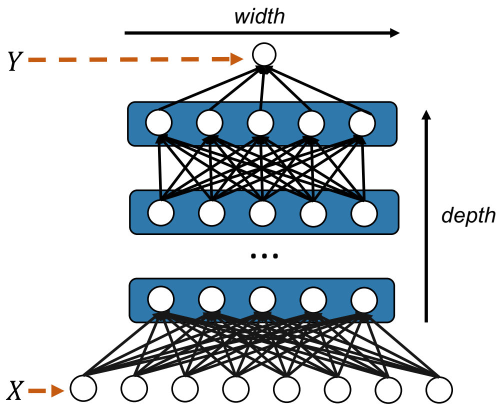


Figure 4.1: Illustration of a Deep Neural Network: The input consists of an input vector X. This is then fed into a network with n hidden layers, which then makes a prediction for the cardinality of the query, Y.

with weights, θ. Each layer contains a collection of neurons, which help express non-linearity. During training, the behavior of the inner layers are not defined by the input data X, instead these models must learn how to tune the weights to produce the correct output. Since there is no direct interaction between the layers and the input training data, these layers are called *hidden layers* [52].

Training occurs through a series of steps. First, during forward propagation, a fixed-sized input X is fed into the network through the input layer. This input is propagated through each layer through a series of weights [119] until it reaches the final output, Y. This process is illustrated in Figure 4.1. After a forward pass, the backpropagation step then evaluates the error of the network and through gradient descent, modifies the weights to improve errors for future predictions.

There are several architectures we could consider for the model. As shown in Figure 4.1, a neural network can have a different number of layers (depth) and a different number of hidden units in each individual layer (width). Determining the correct number of hidden units is currently an active area of research and does not have strong theoretical principles [52]. Although a network with only a single wide hidden layer is capable of learning a complex function, deep networks are able to use a smaller number of training parameters to achieve the same goal. Unfortunately, deep

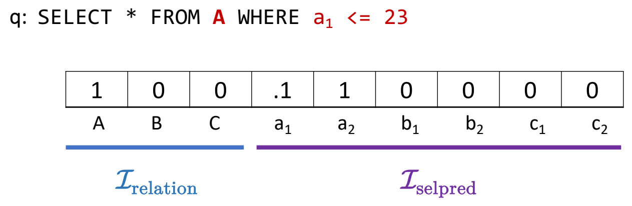

q: SELECT * FROM A WHERE a_1 <= 23
1
0
0
.1
1
0
0
0
0
A
B
C
a_1
a_2
b_1
b_2
c_1
c_2
$\mathcal{I}_{\text{relation}}$
$\mathcal{I}_{\text{selpred}}$

Figure 4.2: Query Encoding for selections: We encode a selection query by specifying the underlying relations and all selection predicate values.

networks are difficult to train and to optimize [52]. In this work, we focus on evaluating a variety of network architectures. We focus on simple architectures comprising a small number of fully connected layers. We vary the width and the depth of the network. More complex architectures are possible [69] and are also interesting to study. Our goal, however, is to understand the performance of basic architectures first.

Given a model, a query q and a fixed dataset D, we define an encoding for the input, X. The input X should contain enough information for the model to learn a useful mapping. There are several ways to represent a query as an input vector. The encoding determines how much information we provide the network. In this work, we define X as a concatenation of three single dimensional vectors: $\mathcal{I}_{relations}$, $\mathcal{I}_{selpred}$, and $\mathcal{I}_{joinpred}$. To explain this encoding, we first describe how to encode selection queries.

Modeling Selection Queries

With selections queries, the goal is to have the network learn the distribution of each column and combinations of columns for a single relation. To encode a selection query, we provide the model with information about *which* relation in D we are applying the selections to, along with the attribute values used in the selection predicates. We encode the relation using vector $\mathcal{I}_{relations}$, and a binary one-hot encoding. Each element in $\mathcal{I}_{relations}$ represents a relation in D. If a relation is referenced in q, we set the designated element to 1, otherwise we set it to 0.

We encode the selection predicates in q using vector $\mathcal{I}_{selpred}$. As described in Section 4.1, selection predicates are limited to single-sided range predicates. Each element in this vector holds

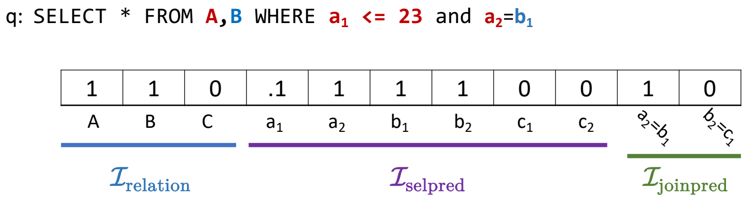

q: SELECT * FROM A,B WHERE a1 <= 23 and a2=b1
1
1
0
.1
1
1
1
0
0
1
0
A
B
C
a1
a2
b1
b2
c1
c2
a2=b1
b2=c1
$\mathcal{I}_{\text{relation}}$
$\mathcal{I}_{\text{selpred}}$
$\mathcal{I}_{\text{joinpred}}$

Figure 4.3: Query Encoding for Joins+Selections: We encode a join+selection query based on the joined relations, the selections predicate values and join predicates.

the selection value for one attribute. The vector includes one element for each attribute of each relation in D. As an example, assume we have the following query: SELECT * FROM A WHERE $a_1 \leq 23$. In this case, we set the corresponding element for a_1 in $\mathcal{I}_{selpred}$ as 23. Otherwise, if there is no selection on an attribute, we set the element with the maximum value of the attribute's domain. This captures the fact that we are selecting all values for that attribute.

Neural networks are highly sensitive to the domain of the input values. Having unnormalized values in the input will highly impact the error surface for gradient descent, making the model difficult to train. Instead, we encode these selection predicates as values ranging from 0 to 1, where the value represents the percentile of the attribute's active domain as shown in Figure 4.2. The output of the model is also normalized to represent the percentage of tuples that remain after the selection query is applied to the relation. Using this type of normalization, instead of learning query cardinalities, we are learning in fact predicate selectivities.

Modeling Join Queries Introducing queries that contain both joins and selections requires the model to learn a more complex operation. Joins essentially apply a cartesian product across a set of relations followed by additional filters that correspond to the equality join and selection predicates. As we later show in our measurement analysis, deeper networks are usually more successful at predicting join cardinalities than shallower networks. We encode existing *join predicates* with the vector $\mathcal{I}_{joinpred}$ using a binary one-hot encoding. As we now include joins, the output Y now represents the fraction of tuples selected from the join result. Hence, once again, the model will learn the selectivity of the join operation.

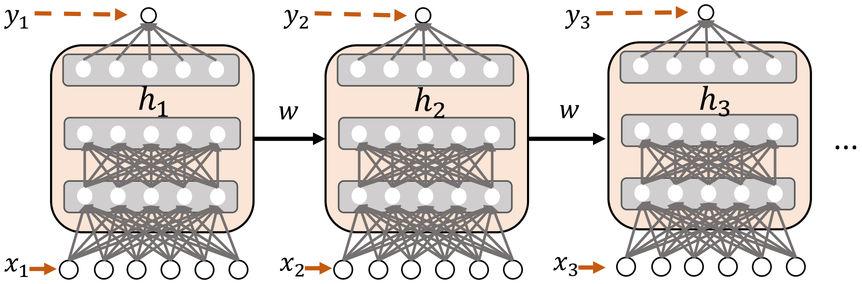
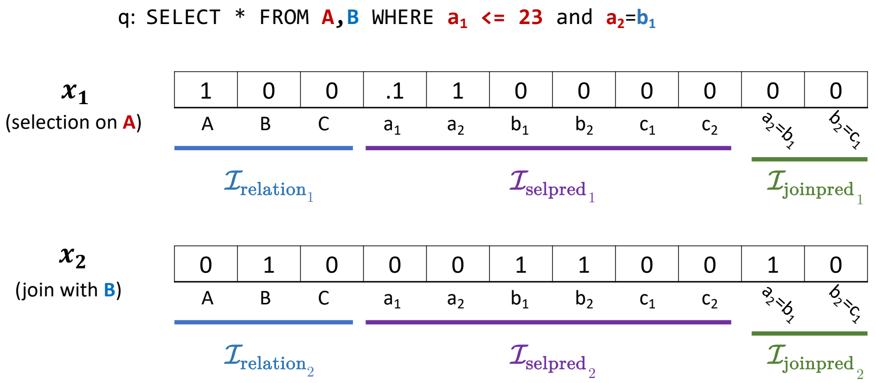


Figure 4.4: Illustration of a Recurrent Neural Network: The input consists of a sequence of inputs $\{x_1, x_2, ..., x_t\}$. Each input, along with the hidden state of the previous timestep, is fed into the network to make a prediction, y_i.

We illustrate this encoding using an example in Figure 4.3. Given the following query from our running example dataset, SELECT $*$ FROM A, B WHERE $a_1 \leq 23$ and $a_2 = b_1$, we show the encoding in the figure. For $\mathcal{I}_{relations}$, there are three possible elements, one for each relation in D. For this query, we only set the elements corresponding to relations A and B to 1. The vector $\mathcal{I}_{selpred}$ contains the encoding for the selection predicates. Since relation C is not referenced in q, we set all of its attributes in $\mathcal{I}_{selpred}$ to 0. We set to $.1$ the element corresponding to attribute a_1, as it represents the percentile of the active domain for a_1. The rest of the attributes from A and B are set to 1, as we are not filtering any values from these attributes. Finally, the vector $\mathcal{I}_{joinpred}$ encodes the join predicate $a_2 = b_1$ with a 1.

4.2.2 Recurrent Neural Networks

As we described in our prior work [94], if we focus on left-deep plans, we can model queries as sequence of operations, and we can leverage that structure when learning a model. Recurrent neural networks (RNN) in particular are designed for sequential data such as time-series data or text sequences [119]. Compared to neural networks where the input is a single vector X, the input to RNNs is a sequence with t timesteps, $X = \{x_1, x_2, ..., x_t\}$. For each timestep t, the model receives two inputs: x_t and h_{t-1}, where h_{t-1} is the generated hidden state from the previous



q: SELECT * FROM A,B WHERE a_1 <= 23 and a_2=b_1

	A	B	C	a_1	a_2	b_1	b_2	c_1	c_2	a_2=b_1	b_2=c_1
x_1 (selection on A)	1	0	0	.1	1	0	0	0	0	0	0
x_2 (join with B)	0	1	0	0	0	1	1	0	0	1	0

$\mathcal{I}_{\text{relation}_1}$
$\mathcal{I}_{\text{selpred}_1}$
$\mathcal{I}_{\text{joinpred}_1}$
$\mathcal{I}_{\text{relation}_2}$
$\mathcal{I}_{\text{selpred}_2}$
$\mathcal{I}_{\text{joinpred}_2}$

Figure 4.5: Query Encoding for the RNN model: In this example, the input consists of two inputs: $\{x_1, x_2\}$. The first input represents the subquery that scans and filters relation A. The second represents the join with relation B.

timestep [119]. With these inputs, the model generates a hidden state for the current timestep t, where $h_t = f(h_{t-1}, x_t)$ and f represents an activation function. Given this feedback loop, each hidden state contains traces of not only the previous timestep, but all those preceding it as well. RNNs can either have a single output Y for the final timestep (a many-to-one architecture) or they can have one output for each timestep (many-to-many) where $Y = \{y_1, y_2, ..., y_t\}$.

In our context, we model queries as a series of actions, where each action represents a query operation (i.e. a selection or a join) in a left-deep query plan corresponding to the query. With a sequential input, RNNs incrementally generate succinct representations for each timestep, which are known as *hidden states*, and which represent a subquery. Recurrent neural networks rely on these hidden states to infer context from previous timesteps. More importantly, h_t is not a manually specified feature vector, but it is the latent representation that the model learns itself. We illustrate how these hidden states are generated in Figure 4.4. Information from each hidden state, h_t is fed into the next timestep, $t + 1$ through shared weights, w. In our context, hidden representations are useful, as they capture important details about the underlying intermediate result. The information learned at the hidden state is highly dependent on the input and output of the network.

To generate the input for this model, we concatenate three input vectors for *each* action x_i.

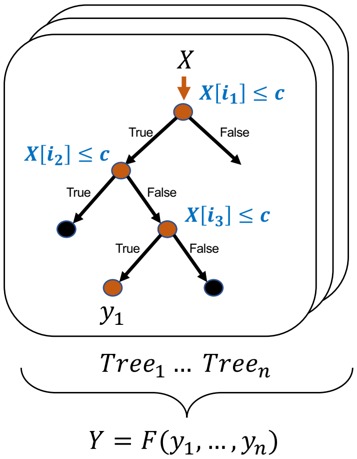


Figure 4.6: Illustration of a Random Forest Model: Each input X is tested against each criteria in each node (where each $X[i]$ represents an attribute in X). The predictions from each tree are aggregated for the final prediction, Y.

That is, for each action x_i, we concatenate vectors $\mathcal{I}_{relation_i}$, $\mathcal{I}_{selpred_i}$ and $\mathcal{I}_{joinpred_i}$. In Figure 4.5, we show the representation for our running example, SELECT $*$ FROM A, B WHERE $a_1 \leq 23$ and $a_2 = b_1$. We break down this query into two operations: the scan and selection on relation A, followed by a join with relation B. Alternatively, we could have the first action represent the scan on relation B (with no selections applied), followed by a join and selection with relation A.

4.2.3 Random Forest

In this study, we also include random forest models, as they are fast to build and are well suited for regression and classification problems [35]. A random forest model is a combination of predictions from several independently trained trees as shown in Figure 4.6. Each tree, $Tree_i$, is a sequence of decisions that leads to a prediction at the leaf nodes, y_i. With each tree, the data is repeatedly split based on different attributes from X. Each tree partitions the input space into subregions, where each of these subregions either contains a linear model [107] or a constant to make a prediction [22]. Finding these subregions requires finding an optimal set of splits, which is computationally infeasible. Instead, these models use a greedy optimization to incrementally grow the trees one node at a time. To help generalize the model and to help add randomness, the random forest model uses a bootstrapped dataset from the training data to build many trees [41]. Based on the predictions from all these trees, the random forest model uses a function F (usually the mean) to compute a final prediction Y.

4.3 Measurement Analysis

In this section, we evaluate all models and their architecture variants on three datasets. We start with a description of the experimental setup, which includes the implementation of the models and generation of the training datasets. We then evaluate the accuracy, time, space trade-offs in subsection 4.3.2, followed by a study of the robustness of the models in subsection 4.3.3 and a look into their latents in subsection 4.3.4.

4.3.1 Experimental Setup

Datasets: We evaluate the models on three datasets:

- **IMDB**: The *Internet Movie Data Base* is a real dataset that contains a wide variety of information about actors, movies, companies, etc. This dataset has 21 relations. The dataset is based on the 3.6GB snapshot from Leis et. al. [77].

- **DMV**: This dataset contains 6 relations (61MB) and is based on a real-world dataset from a Department of Motor Vehicles [68]. Relations include accidents, owners, cars, location, demographics and time.

- **TPC-H (skewed)** : This is a standard benchmark dataset with 8 relations and a scale factor of 1 (1GB). We adjust the skew factor to $z = 1$ [31].

Model Architectures: For the recurrent and neural networks, we build several models that vary in width (w) and depth (d). To minimize the number of possible architecture combinations, we assume that all layers within a model have the same width. We annotate the models as a pair (x, y), where x represents the width and y represents the depth. For example, (100w, 1d) represents a model with 100 hidden units in a single hidden layer. For the random forest models, we vary the number of trees from 1 to 500. No additional pruning takes place. We compare these models to estimates from PostgreSQL version 9.6 [105]. To fairly compare PostgreSQL to these models in terms of space, we modify the PostgreSQL source to allow for a larger number of bins in each histogram. For each relation in each dataset, we collect statistics from each join predicate column and each selection column. We vary the number of bins from the default size (100 bins) up to 100K.

Training Data: For each dataset, we generate various training sets with different levels of query complexity. We define three complexity levels: $2Join$, $4Join$ and $6Join$. $2Join$ is the case where we generate a training set with joins that consist of any 2 relations in the dataset, $4Join$ represents joins with 4 relations, and $6Join$ represents joins with 6 relations. In addition, for each dataset, we manually select a set of columns as candidates for selection predicates. We select columns with small discrete domain sizes, as these are generally the columns that contain more semantically meaningful information about the data, unlike columns that contain a sequence of identifiers. As we generate the workload, selection predicate values are randomly drawn from the domains of the selected candidate columns. We generate 100K training samples for each query complexity and each dataset. We randomly select the desired number of tables and pick the selection columns from the joined relations. For the RNN, because it requires an input for each timestep, we extend these training sets by adding more training samples for all the subqueries. For example, for a query that joins six relations, we extend the training set with additional examples representing the subquery after each intermediate join. When generating the subqueries, we randomly select the order of the join operations. For each query complexity training set, we select 1K samples to serve as the test set.

Hyperparameter Tuning: We tune each model architecture for each dataset. We separate 10% of the training data as the validation data. We run a basic grid search over the learning rate and batch size. A larger batch size (although faster to train, especially on a GPU) might lead to sub-optimal results, while a small batch size is more susceptible to noise. Larger learning rates also have the tendency to oscillate around the optimum, while smaller learning rates might take a long time to train. We set the number of epochs to 500 for all learning rate and batch parameter combinations. Based on the combination that leads to the lowest learning rate, we continue to train for more epochs as long as the validation loss keeps decreasing. We stop the training once the validation loss plateaus or increases.

Model Implementation Details: The neural network is implemented in Tensorflow [5] and is implemented as a residual network with leaky RELU activation functions, as it is a default recommendation to use in modern neural networks [52]. Weights are initialized from a random normal distribution with a small standard deviation. Biases are initialized to .01. The input X is normalized as explained in subsection 4.2.1 and centered using a StandardScaler. The output Y is log transformed and also normalized with a StandardScaler. The model's goal is to minimize the mean squared error between the real outputs and the predictions. We use the AdamOptimizer as the optimizer for the model. The recurrent neural network is also implemented in Tensorflow. For deep recurrent neural networks, we use a ResidualWrapper around each layer, to mimic the residual implementation of the neural networks. Both the neural network and recurrent neural networks are run on a GPU on p2.xlarge instances on Amazon AWS [11]. Finally, the Random Forest model is based on an implementation from sklearn's RandomForestRegressor module [101].

4.3.2 Learning Cardinalities for Selections + Joins

In this section, we vary the architecture of the models and evaluate them on the three datasets. We study the trade-offs (space, time, and accuracy) for these models.

First, we evaluate the prediction accuracy for each model. As described in Section 4.1, we make the assumption that the query workload is known in advance (we relax this assumption later in this section). In this case, the models overfit to a specific set of queries. For each query complexity, we train six neural network (NN) and six recurrent neural network models (RNN) based on the following widths and depths: (100w, 1d), (100w, 5d), (500w, 1d), (500w, 5d), (1000w, 1d), (1000w, 5d). We separately train four random forest models (RF) with 1, 5, 50 or 500 trees. Larger models generally use up more space, but result in more accurate cardinality predictions.

To make this analysis comparable to PostgreSQL, we first limit the storage budget for the models to be no more than the storage budget for PostgreSQL histograms. We compute the size of a model as the size of all its parameters. For the NN and RNN models, we thus measure the number of trainable variables and for PostgreSQL, we measure the number of parameters used in the pg_stats table. We compare PostgreSQL cardinality estimates to those produced by models that are smaller in size compared to PostgreSQL's histograms. We specifically study the PostgreSQL scenario where each histograms builds at most 1K bins. Setting the number of bins to 1K for PostgreSQL results in 13385 parameters for the DMV dataset, 15182 parameters for the TPC-H dataset and 44728 parameters for the IMDB dataset. We purposely set PostgreSQL as the storage upper bound size. Given these storage budgets, we then select the best neural network architecture, the best recurrent network architecture, and the best random forest model. If no model meets the budget, we do not display them on the graphs. If more than one model architecture meets the storage budget, we display the best model, where the best model is defined as the one with the lowest median error.

Limited Storage CDFs and Outlier Analysis: For PostgreSQL, as for other relational DBMSs, cardinality estimation is easy for some queries and hard for others. As expected PostgreSQL yields more accurate predictions for queries with a low complexity, particularly those with no selection predicates. To help distinguish between these "easy" and "hard" queries (labeled as Easy(PostgreSQL) and Hard(PostgreSQL)), we plot the absolute errors from PostgreSQL as a cumulative distribution (cdf) as shown in Figure 4.7. We use the knee (k) of the cdf curve to split the queries into an "easy" category (those with errors less than the knee, k) and a "hard" category (those with errors greater than the knee k). We compute the k through the Kneedle algorithm, which returns the point of maximum curvature [114].For the TPC-H dataset, the distribution of errors is wide. To ensure we retain enough queries in the Hard(PostgreSQL) category (for later more in-depth analysis), we compute k and half the corresponding error.

We first focus on the Hard(PostgreSQL) queries. These are the more interesting queries to study as these are the queries for which we seek to improve cardinality estimates. We plot the

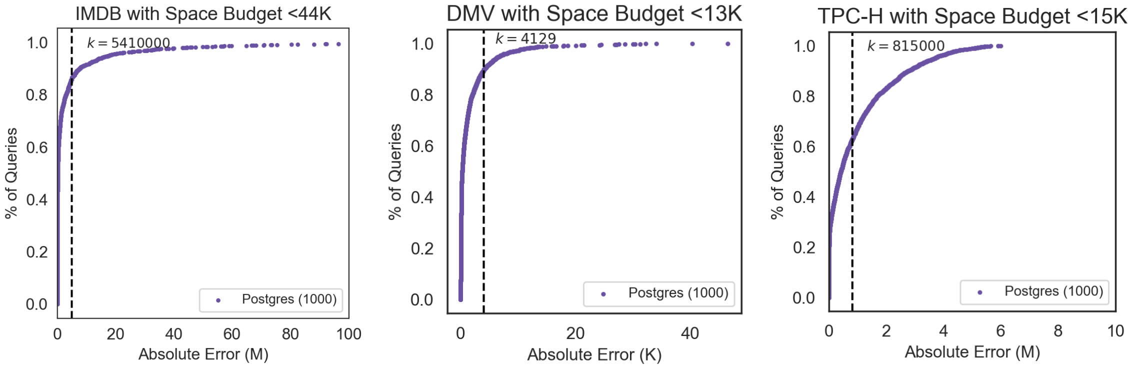
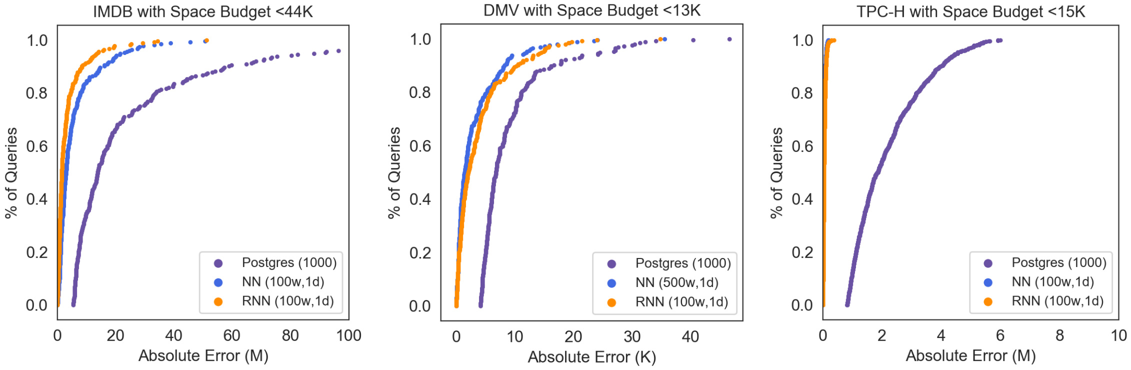


Figure 4.7: CDF of PostgreSQL absolute errors with storage budget: For each curve, we show the knee, k, which defines the split between Easy(PostgreSQL) and Hard(PostgreSQL). The x-axis shows the absolute error, where K and M represent the scale in thousands and millions respectively.

distribution of errors for the best performing models for each dataset in Figure 4.8. Overall, both types of models outperform PostgreSQL on all three datasets. We also find the performance of both types of models to be similar.

First, in Figure 4.8a, we show the cdf for the Hard(PostgreSQL) queries from the IMDB dataset. From the entire set of IMDB queries, 11% of the queries fall in the Hard(PostgreSQL) category. The y-axis represents the percentage of queries and the x-axis represents the absolute error. In addition to the PostgreSQL error curve, we show the cdf for the corresponding queries from the best neural network and recurrent neural network models. We do not show the random forest models here, as the smallest model does not meet the storage budget. Both the neural network and recurrent neural network have comparable cardinality estimation errors. On average, the neural network reduces estimation error by 72%, while the recurrent neural network reduces the error by 80%. Below Figure 4.8a, we include additional details that show the percentiles of the model cdfs, the average absolute error for each query complexity, and the average relative error.

In Figure 4.8b, we show the cdf for Hard(PostgreSQL) from the DMV dataset. Approximately 10% of the queries are labeled as hard for PostgreSQL. The NN reduces the errors by 75% on



(a) IMDB with Storage Budget (b) DMV with Storage Budget (c) TPCH with Storage Budget

	CDF Percentiles		
	25%	50%	75%
PostgreSQL	7.8M	13.8M	29.7M
NN (100w,1d)	1.25M	2.97M	6.6M
RNN (100w,1d)	.71M	1.49M	3.67M

	CDF Percentiles		
	25%	50%	75%
PostgreSQL	5.2K	6.8K	10.3K
NN (500w,1d)	.50K	1.4K	4.1K
RNN (100w,1d)	.76K	2.1K	5.1K

	CDF Percentiles		
	25%	50%	75%
PostgreSQL	1.2M	1.8M	2.9M
NN (100w,1d)	.01M	.02M	.05M
RNN (100w,1d)	.01M	.03M	.06M

	Average Absolute Errors		
	$2Join$	$4Join$	$6Join$
PostgreSQL	6.8M	12.8M	31.3M
NN (100w,1d)	.80M	4.1M	7.0M
RNN (100w,1d)	.58M	2.2M	4.1M

	Average Absolute Errors		
	$2Join$	$4Join$	$6Join$
PostgreSQL	9.4K	9.4K	8.9K
NN (100w,1d)	4.9K	3.0K	3.1K
RNN (100w,1d)	7.9K	2.9K	4.0K

	Average Absolute Errors		
	$2Join$	$4Join$	$6Join$
PostgreSQL	2.9M	2.2M	1.8M
NN (100w,1d)	35K	40K	32K
RNN (100w,1d)	27K	60K	41K

	Average Relative Errors		
	$2Join$	$4Join$	$6Join$
PostgreSQL	.39	.75	.95
NN (100w,1d)	.04	.22	.20
RNN (100w,1d)	.03	.11	.13

	Average Relative Errors		
	$2Join$	$4Join$	$6Join$
PostgreSQL	.10	.20	.23
NN (100w,1d)	.03	.03	.04
RNN (100w,1d)	.06	.02	.07

	Average Relative Errors		
	$2Join$	$4Join$	$6Join$
PostgreSQL	.99	.99	.99
NN (100w,1d)	.01	.02	.01
RNN (100w,1d)	.01	.03	.02

Figure 4.8: Error Analysis for all Models : We show the curve for Hard(PostgreSQL) and show the corresponding errors from the best models below the storage budget. Below each graph, we show tables detailing the percentiles, the average absolute error and average relative error.

average and the RNN by 73%. As shown in the tables below the figure, the complexity of the queries does not heavily impact the average error. In fact, the relative errors for the NN across all query complexities have a small standard deviation ($\sigma = .004$), compared to IMDB ($\sigma = .08$). Compared to DMV, the IMDB dataset contains several many-to-many primary/foreign key relationships, so joining relations significantly increases the size of the final join result.

	% Queries Easy **(Models)**					
	IMDB		DMV		TPC-H	
	NN(100w,1d)	RNN(100w,1d)	NN(500w,1d)	RNN(100w,1d)	NN(100w,1d)	RNN(100w,1d
Easy(PostgreSQL)	99.5%	99.8%	90.5%	94.5%	100%	100%
Hard(PostgreSQL)	71.4%	83.5%	75.6%	69.4%	100%	100%

Table 4.1: Percentage of Queries that are Easy for the Models: For each Easy(PostgreSQL) query batch, we find the percentage of queries that are also easy for the models. We also show the percentage of queries that are easy based on the Hard(PostgreSQL) batch

We also observe a significant error reduction in Figure 4.8c (TPC-H), where the NN improves estimates by 98% and the RNN by 97%. For TPC-H, 30% of the queries are hard for PostgreSQL.

Table 4.1 shows the percentage of queries that are easy for the models given that they are either Easy(PostgreSQL) or Hard(PostgreSQL) for PostgreSQL. In the case of Hard(PostgreSQL) queries, 70% or more become easy with the models. For the Easy(PostgreSQL) queries, the simple NN and RNN models also find a majority of these queries to be easy (¿90%). For IMDB and DMV, there are some queries from the Easy(PostgreSQL) batch that the models find to be hard. We highlight some of these hard queries below:

- From the IMDB dataset, approximately 0.4% of the Easy(PostgreSQL) queries are hard for the NN, and we find that the query with the highest error is one with an absolute error of 8.9M. This query joins the $name$, $cast\_info$, $role\_type$, and $char\_name$ relations and has a selection predicate on $role\_id <= 8$. For the RNN, the query with the highest error is similar. It joins the $name$, $cast\_info$, $role\_type$, and $char\_name$ relations, with a selection predicate on $role\_id <= 4$.

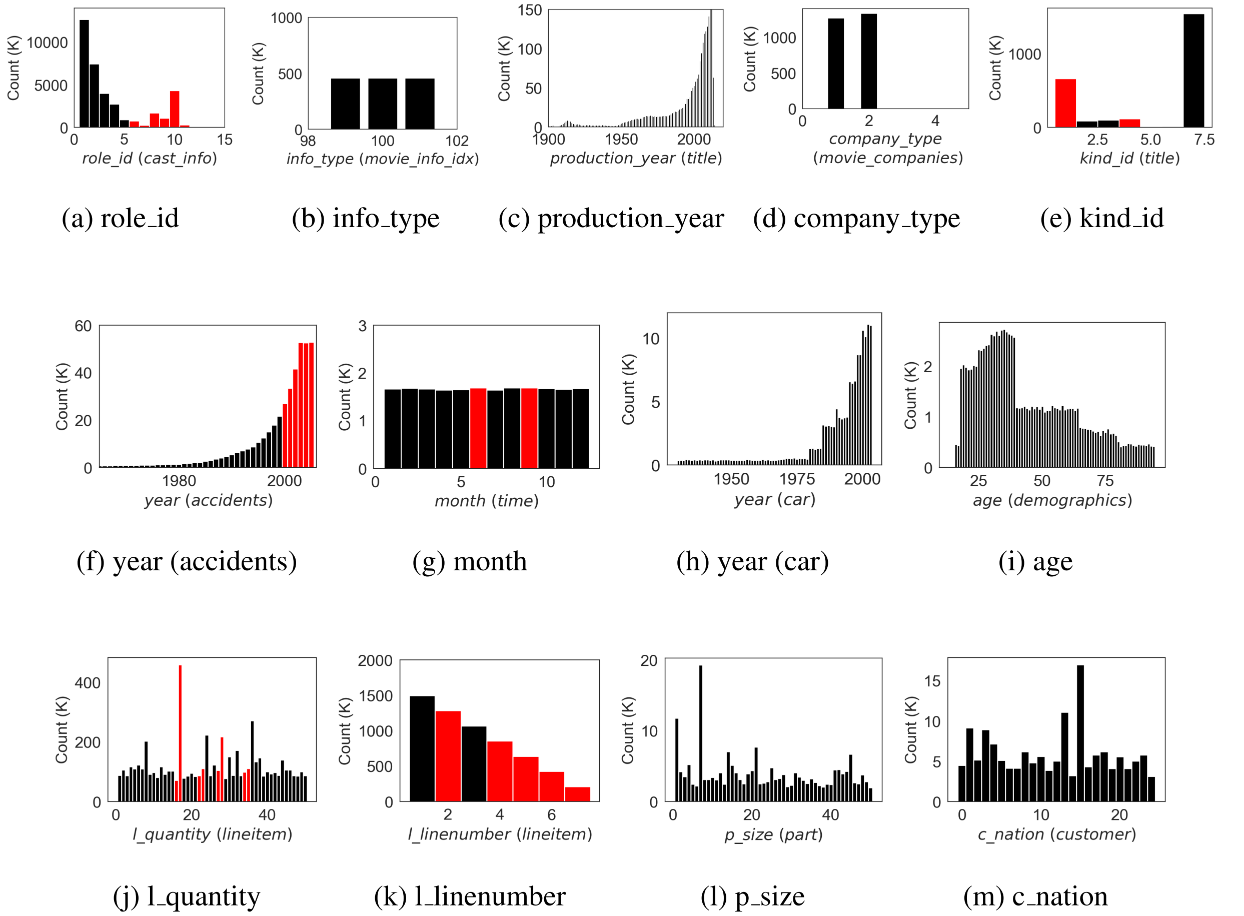
- For the DMV dataset, approximately 10.5% of the queries are hard for NN, and 6.5% are hard for the RNN. The query with the highest error for the NN is one that joins all relations car, $demographics$, $location$, $time$, $owner$, and $accidents$ and has several selection predicates:

$age\_demographics <= 89$, $month\_time <= 12$, $year\_accidents <= 2004$. For the RNN, the query with the highest absolute error also joins all relations and has selection predicates with similar values, $age\_demographics <= 93$,$month\_time <= 9$,$year\_accidents <= 2005$.

	Queries with Highest Errors from $2Join$	
	Best NN per Dataset	**Best RNN per Dataset**
IMDB	(cast_info,role_type) where role_id<= 11 [1.9M]	(cast_info,name) where role_id <= 9 [2.1M]
	(cast_info,role_type) where role_id <= 10 [1.9M]	(cast_info,name) where role_id<= 11 [2.1M]
	(cast_info,role_type) where role_id <= 8 [1.9M]	(cast_info,role_type) where role_id <= 11 [1.9M]
	(cast_info,title) where kind_id <= 1,production_year <=\ 2019,role_id<= 4 [1.5M]	(cast_info,role_type) where role_id <= 9 [1.8M]
	(movie_info,info_type) [1.3M]	(cast_info,role_type) where role_id <= 7 [1.3M]
	(cast_info,role_type) where role_id <= 7 [1.2M]	(cast_info,name) where role_id <= 8 [1.2M]
	(cast_info,role_type) where role_id <= 6 [1.1M]	(cast_info,role_type) where role_id <= 7 [1.1M]
	(cast_info,title) where kind_id <= 4,production_year <=\ 2019,role_id<= 6 [1.0M]	(cast_info,title) where kind_id <= 4,production_year <=\ 2019,role_id<= 6 [.9M]
DMV	(accidents,time) where year<= 2005 and month <= 9 [11K]	(accidents,time) where year <= 2005 and month <= 9 [33K]
	(accidents,time) where month <= 9 and year <= 2005 [11K]	(accidents,location) where year <= 2003 [18K]
	(accidents,time) where year <= 2003 and month <= 6 [10K]	(car,accidents) where year <= 2005 [18K]
	(accidents,time) where year <= 2000 and month <= 6 [10K]	(car,accidents) where year <= 2003 [18K]
	(accidents,location) where year <= 2001 [7K]	(accidents,time) where year <= 2005 and month <= 9 [17K]
	(accidents,location) where year <= 2000 [7K]	(accidents,time) where year <= 2003 and month <= 6 [10K]
	(car,accidents) where year <= 2005 [6K]	(accidents,time) where year <= 2000 and month <= 6 [10K]
	(car,accidents) where year <= 2004 [6K]	(accidents,location) where year <= 2002 [8K]
TPC-H	(lineitem,orders) l_linenumber <= 7 and l_quantity <=16 [116K]	(lineitem,orders) where l_linenumber <= 7 and l_quantity <= 34 [89K]
	(lineitem,orders) l_linenumber <= 6 and l_quantity<= 16 [109K]	(lineitem,orders) where l_linenumber<= 6 and l_quantity <= 38 [75K]
	(lineitem,orders) l_linenumber <= 5 and l_quantity <= 35 [98K]	(lineitem,orders) where l_linenumber <= 5 and l_quantity <= 35 [72K]
	(lineitem,orders) l_linenumber <= 7 and l_quantity <= 34 [84K]	(lineitem,orders) where l_linenumber <= 6 and l_quantity <= 28 [62K]
	(lineitem,orders) l_linenumber <= 7 and l_quantity <= 27 [65K]	(lineitem,orders) where l_linenumber <= 7 and l_quantity <= 22 [55K]
	(lineitem,orders) l_linenumber <= 2 and l_quantity <= 17 [61K]	(lineitem,orders) where l_linenumber <= 5 and l_quantity <= 28 [52K]
	(lineitem,orders) l_linenumber <= 6 and l_quantity <= 27 [61K]	(lineitem,orders) where l_linenumber <= 4 and l_quantity <= 28 [51K]
	(lineitem,orders) l_linenumber <= 7 and l_quantity <= 23 [59K]	(lineitem,orders) where l_linenumber <= 7 and l_quantity <= 23 [50K]

Table 4.2: The $2Join$ Queries with the Highest Errors from the NN and RNN Models: For each dataset, we show the top eight queries with the highest absolute errors.

From the Hard(PostgreSQL) queries, there are more queries that remain difficult for the models compared to Easy(PostgreSQL). These hard queries consist of joins of 6 relations (the most



(a) role_id (b) info_type (c) production_year (d) company_type (e) kind_id

(f) year (accidents) (g) month (h) year (car) (i) age

(j) l_quantity (k) l_linenumber (l) p_size (m) c_nation

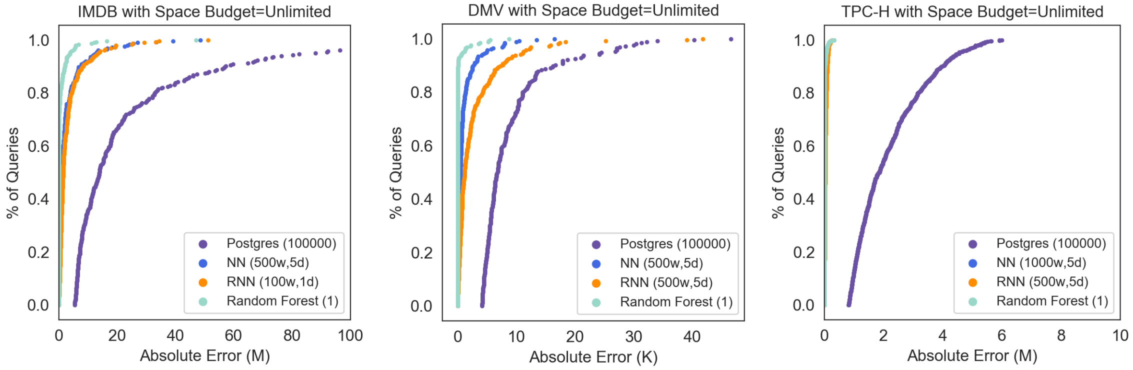
Figure 4.9: Distributions for all Selection Columns: First row shows all distributions from the IMDB relation. Second shows distributions from DMV, and the third shows TPC-H.

complex queries we have in the test set) and up to 5 selection predicates.

Understanding why the NN or RNN fail to accurately predict the cardinality for specific queries is challenging as there are several factors to consider. For example, the error could be caused by a specific join or perhaps a combination of selection attributes. To gain a better understanding of these errors, we now only focus on the queries with a low complexity (i.e. those from the $2Join$ test set). In Table 4.2, we take the Easy(PostgreSQL) queries and show the queries with the highest errors from the $2Join$ set. For succinctness, we annotate each query with the names of the relations it joins (relations are listed in parenthesis) and its selection predicates. We further add the absolute error of the query in brackets.

For IMDB, the hardest queries for the NN and RNN are similar. These queries consist of joins with $cast\_info$ and either $role\_type$ or $name$. All queries also have a selection predicate on the $role\_id$ column with values between 6 and 11. Figure 4.9 shows the value distributions for different attributes. The first row shows all selection columns for IMDB, the second for DMV, and third for TPC-H. The x-axis in each graph represents the column value and the y-axis represents the frequency of the value. In Figure 4.9a, we show the distribution of the $role\_id$ column. The red bars represent the values for which we see the highest errors for the NN and RNN models, based on Table 4.2. Compared to the other existing selection attributes, $role\_id$ comes from the largest relation in the dataset, $cast\_info$. We generally observe that the models have the highest errors for columns that belong to the largest relations and specifically at the points where the distribution is irregular.

For the DMV dataset, the hardest queries are those that contain the $accidents$ relation and join with $time$ or $location$. These queries have selection predicates on the $year$ and $month$ columns. We highlight the selection predicate values in Figure 4.9f and Figure 4.9g. We note that there is a one-to-one mapping between the $accidents$ and $time$ relation, so the distribution for these columns does not change due to the join. This is also the case for the join between $accidents$ and $location$. The year column in the accidents relation has a high skew and the models have the highest errors for the more frequent values. The accidents relation also happens to be the largest relation in the DMV dataset.



(a) IMDB Unlimited

(b) DMV Unlimited

(c) TPCH Unlimited

CDF Percentiles			
	25%	50%	75%
PostgreSQL	7.8M	13.9M	27.5M
NN (500w,5d)	.30M	.97M	2.6M
RNN (100w,1d)	.6M	1.4M	3.6M
Random Forest (1)	1e-6M	6e-6M	.2M

CDF Percentiles			
	25%	50%	75%
PostgreSQL	5.28K	6.82K	10.3K
NN (500w,5d)	.16K	.46K	1.0K
RNN (500w,5d)	.02K	1.1K	3.2K
Random Forest (1)	3e-6K	6e-6K	1e-5K

CDF Percentiles			
	25%	50%	75%
PostgreSQL	1.2M	1.8M	2.9M
NN (1000w,5d)	.01M	.02M	.04M
RNN (500w,5d)	.01M	.02M	.05M
Random Forest (1)	4e-8M	8e-8M	1e-7M

Average Absolute Errors			
	$2Join$	$4Join$	$6Join$
PostgreSQL	6.8M	12.8M	31.3M
NN (500w,5d)	.28M	.45M	4.1M
RNN (100w,1d)	.58M	2.0M	4.1M
Random Forest (1)	1e-6M	.21M	1.0M

Average Absolute Errors			
	$2Join$	$4Join$	$6Join$
PostgreSQL	9.5K	9.3K	8.9K
NN (500w,5d)	.4K	.9K	1.3K
RNN (500w,5d)	1.1K	3.3K	2.8K
Random Forest (1)	8e-5	.06K	.29K

Average Absolute Errors			
	$2Join$	$4Join$	$6Join$
PostgreSQL	2.9M	2.2M	1.8M
NN (1000w,5d)	16K	41K	28K
RNN (500w,5d)	29K	59K	23K
Random Forest (1)	9e-4	3K	15K

Average Relative Errors			
	$2Join$	$4Join$	$6Join$
PostgreSQL	.38	.80	.95
NN (500w,5d)	.01	.03	.11
RNN (100w,1d)	.03	.11	.13
Random Forest (1)	3e-8	.01	.03

Average Relative Errors			
	$2Join$	$4Join$	$6Join$
PostgreSQL	.10	.20	.23
NN (500w,5d)	.002	.01	.01
RNN (500w,5d)	.006	.03	.04
Random Forest (1)	7e-8	.0008	.007

Average Relative Errors			
	$2Join$	$4Join$	$6Join$
PostgreSQL	.99	.99	.99
NN (1000w,5d)	.007	.02	.01
RNN (500w,5d)	.01	.02	.01
Random Forest (1)	4e-8	.001	.009

Figure 4.10: Error Analysis for all Models : We show the curve for Hard(PostgreSQL) and show the corresponding errors from the best models with an unlimited storage budget. Below each graph, we show tables detailing the percentiles, the average absolute error and average relative error.

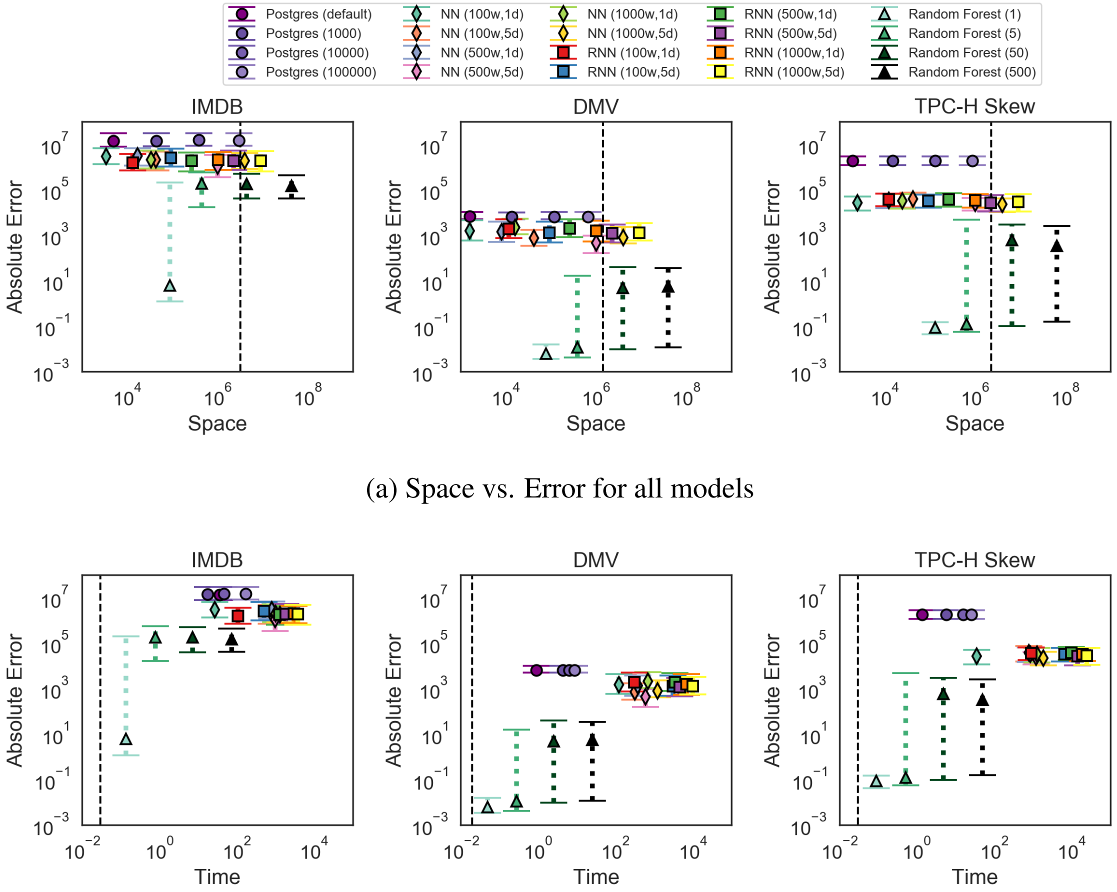
For the TPC-H dataset, most of the errors come from the join between $lineitem$ and $orders$. These contain selection predicates on both the $l\_linenumber$ and $l\_quantity$. The pearson correlation for these two attributes is low (.0002) so these are independent attributes. We highlight the values with highest errors in Figure 4.9j and Figure 4.9k. We note that the $l\_quantity$ in particular has an irregular distribution, and also belongs to the relation with the highest number of tuples in the dataset, $lineitem$.

Unlimited Storage CDFs and Outlier Analysis In Figure 4.10, we show similar graphs across all datasets, but with an unlimited storage budget. The goal here is to understand how more complex models compare against the simpler ones from Figure 4.8. Given this unlimited budget, we now include the random forest models. The PostgreSQL estimates do not significantly change even with 100K bins, which implies that adding finer granularity to the histograms does not significantly improve estimates. Among all the models, the trees have the lowest errors overall across all query complexities and across all datasets.

Time vs Space vs Accuracy Trade-offs In Figure 4.11, we show the error, space and time trade-offs for each model. First, in Figure 4.11a we compare the error and space. On the y-axis, we show the absolute error between the predicted value and the real value on a log scale. On the x-axis we show the space of each model on a log scale. Each point represents the median error and the error bars represent the 25th and 75th percentiles. For all datasets, all variants of PostgreSQL have the highest errors and increasing histogram bin granularity does not significantly improve performance. Neural networks and recurrent neural networks are fairly competitive in terms of absolute error. In Section 4.4, we study whether deeper models actually learn more context about the relations compared to the shallower ones. Models that are deeper are much larger in terms of space, with small error improvement over simpler models.

In Figure 4.11b, we compare the accuracy to the time (in seconds) it takes to train each model. We do not include the time it takes to run the hyperparameter tuning and we do not include the time it takes to run the training queries. We address the latter overhead in Section 4.4.

Given their large sizes, an important question is whether the models improve upon simply keeping the entire query workload in a hash table (with query features as keys and cardinalities as values). To answer this question we plot the overhead of such a hash table. Given that our training data consists of only 100K samples for each query complexity, our goal is to understand whether the deep learning models can actually compress information and still provide a good accuracy. For the hash table model, we assume that each feature for each training example is equivalent to one



(a) Space vs. Error for all models

(b) Time vs Error for all models

Figure 4.11: Trade-offs between Error, Space and Time: We show the absolute error, space and time for each model and for PostreSQL for different number of bins. The horizontal line represents the space and time for the hash table model.

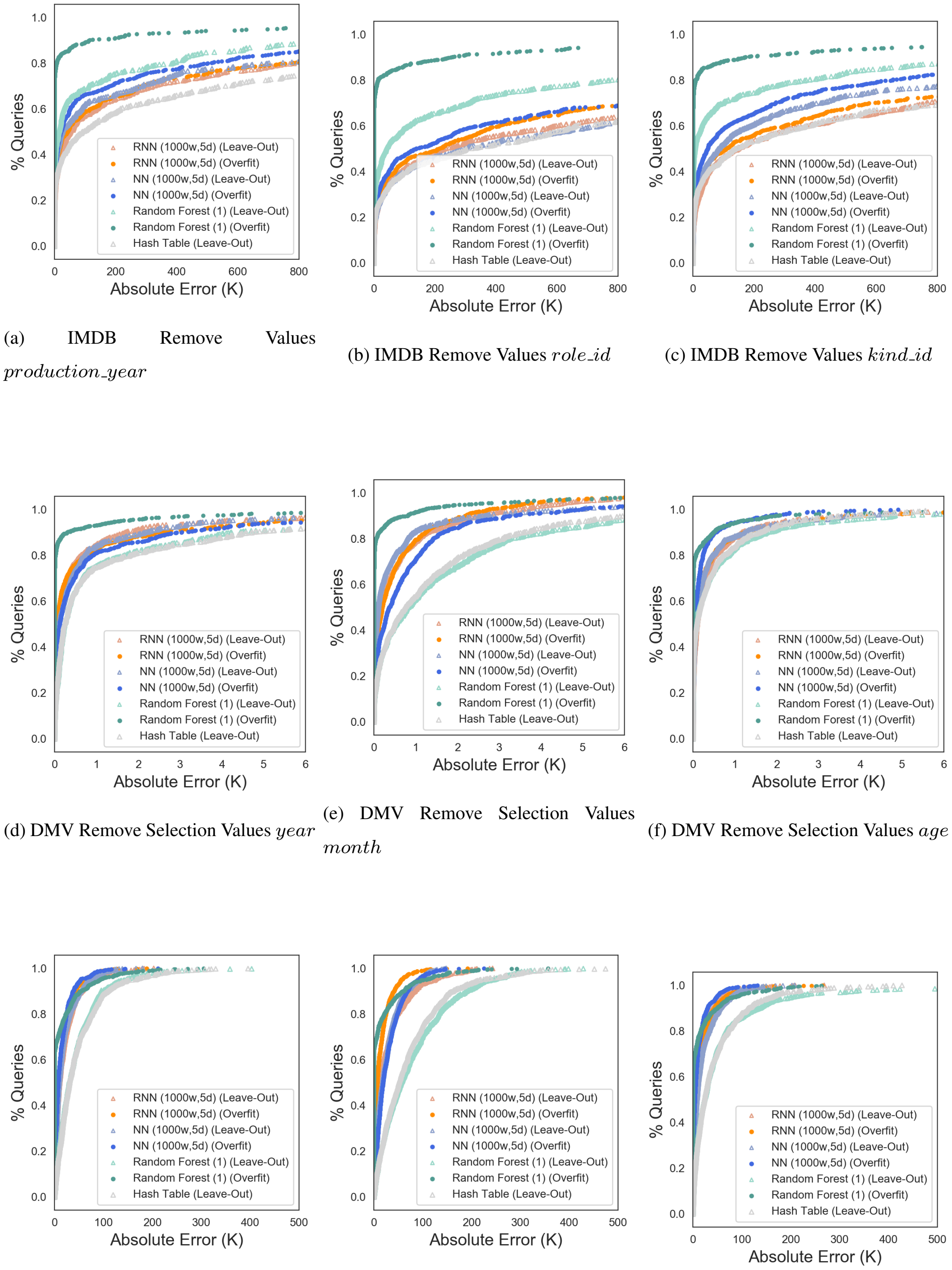
weight when measuring space. To measure time, we measure the time it takes to populate the hash table. We mark this implementation in the graphs as a vertical dashed line. For this model, the error is 0 for each query.

For each dataset, all variants of the tree models result in the lowest error. In particular the trees with the lowest error are those with 1 decision tree. Since we build these models to overfit to a specific query workload, using a single decision tree results in the lowest error. Once more decision trees are introduced, the error is higher as these models no longer overfit and attempt to generalize over the training set. These results suggest that for overfit workloads, the random forest model is able to build these models quickly and more accurately compared to the deep learning models. The deep learning models are able to save in space and although they are not as accurate as the trees, they can still improve errors in some cases by an order of magnitude compared to PostgreSQL.

4.3.3 Model Robustness

In this section, we discuss how sensitive these models are in the face of *unknown* query workloads. That is, instead of overfitting each model to a specific set of queries, we remove some query samples from the training entirely from the $6Join$ sets. We evaluate the most complex models for the RNN and NN (1000w, 5d) as these perform favorably for the $6Join$ set and also select the best performing version of the random forest model as we saw in Figure 4.11 (Random Forest (1)).

Removing Selections In the first row of Figure 4.12, we remove 10% of values from three columns: $production\_year$ in Figure 4.12a, $role\_id$ in Figure 4.12b and $kind\_id$ in Figure 4.12c. In each graph, we have two variants of each model where Leave-Out (shown in lighter colors) is the case where we do not include these queries in the training and Overfit (shown in darker colors) is the case when the models do in fact learn about all possible queries. As shown in Figure 4.12c for the IMDB dataset, the accuracy of the Random Forest model degrades significantly compared to the RNN and NN models for all columns, but still maintains a highest accuracy for all columns. For both the DMV and TPC-H dataset (shown in rows 2 and 3) of Figure 4.12, removing values from training significantly decreases the accuracy for the random forest models. In these graphs, we also included the accuracy of the hash table model (Leave-Out) implementation. Since the samples in the test set are not included in the training set, we use a nearest neighbor approach to find the closest sample that exists in the training set (our hash table). In many cases, the hash model performs similarly to the random forest (Leave-Out) model. For the IMDB dataset, the hash model does not perform as well due to lack of training data. We found that for some queries, the nearest neighbor is actually not very similar. For example, a selection predicate value for the



(a) IMDB Remove Values *production_year*

(b) IMDB Remove Values *role_id*

(c) IMDB Remove Values *kind_id*

(d) DMV Remove Selection Values *year*

(e) DMV Remove Selection Values *month*

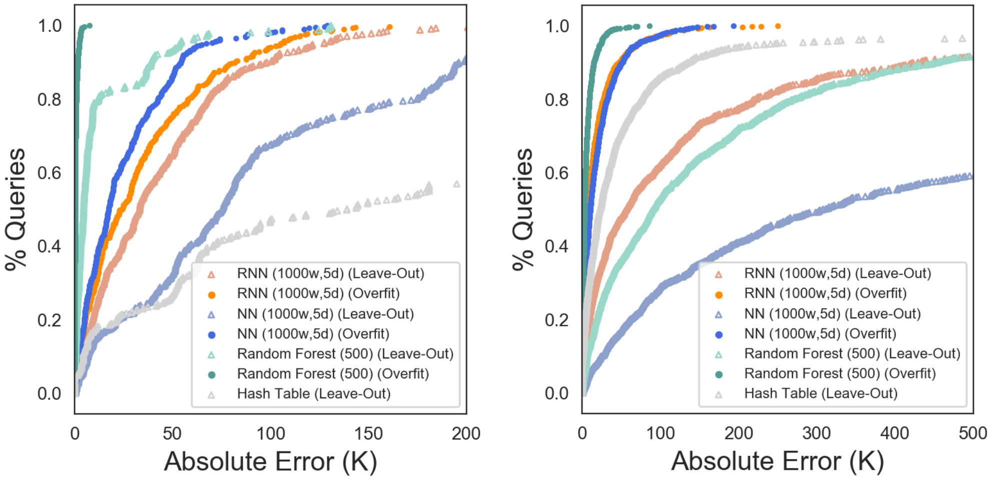
(f) DMV Remove Selection Values *age*

(g) TPC-H Remove Selection *l_quantity*

(h) TPC-H Remove Selection *c_nationkey*

(i) TPC-H Remove Selection *p_size*

Figure 4.12: Removing 10% selection predicate values across all datasets



(a) IMDB Remove Join

(b) TPC-H Remove Join

Figure 4.13: Remove Joins from the Training Workload

nearest neighbor was actually a few values away from the target query.

Removing Joins In Figure 4.13a and Figure 4.13b, we remove a join with a specific combination of tables from the IMDB and TPC-H $6Join$ datasets. During training, the models observe how certain tables join with each other, but they never see the specific combinations we remove. In Figure 4.13a, we remove the join between relations: { $complete\_cast$, aka_title, $movie\_info\_idx$, $title$, $movie\_companies$, $movie\_link$ } from the training set. The queries shown in the figure correspond to the test set, which includes the removed combination of tables with random selection predicates. When comparing between the Overfit and Leave-Out models, there is a large degradation in accuracy for the NN. In particular, the RNN and the random forest model have an advantage over the NN. For the IMDB dataset, we observe that the random forest model relies heavily on features from $\mathcal{I}_{selpred}$. We found that the IMDB dataset contains combinations of tables in the training data that are very similar (and yield the same cardinality) as the combination of tables we removed from the training. In fact, when observing the splits used in the tree, approximately 16% of the splits on average for all test samples are from the features in $\mathcal{I}_{relation}$. Whereas 67% of the splits are from $\mathcal{I}_{selpred}$. It is not clear why the NN model does not learn to rely more on information from $\mathcal{I}_{selpred}$. The hash table model has the worst accuracy, since the nearest neighbor at times selects queries with selection predicates on the same values but different underlying tables.

In Figure 4.13b, we observe a similar trend. For this experiment, we remove a join from the TPC-H dataset with the relations: $\{$ $customer$, $lineitem$, $partsupp$, $nation$, $part$, $orders$ $\}$. Again, the RNN results in lower errors than the neural network, but the tree model still performs better than the RNN and NN. Compared to the IMDB models, the tree model for TPC-H more heavily depends on splits from $\mathcal{I}_{selpred}$. In fact, on average, 88% of the splits are from these selection features. One reason why the random forest might only split on selection features is due to TPC-H's schema. For all the queries in TPC-H's $6Join$ set, the join cardinality (without selections) result in approximately 6M tuples. This implies that regardless of the join combination, the maximum number of tuples is always the same, and the the selections are the defining feature that ultimately determine the final number of tuples.

4.3.4 Model Latents

One challenge of training deep neural networks is the difficulty to understand what the models are actually learning. As discussed in subsection 4.3.3, the random forest models are easily interpretable as we can track path of decision splits to understand how the model is able to predict the outcome given the input. For neural networks, diagnosing why a model arrives at a specific answer is a harder problem. There are several existing approaches, which include masking or altering the input to measure the predication change and studying hidden unit activation values [65, 116, 142].

We study the activation values of the hidden layers for the NN and RNN models. During training, these models take the input, X, and propagate it through a series of transformations that represent the data at different levels of abstraction [52]. Taking a close look at the activation values (also referred to as *latent representations* or *embeddings*) can help diagnose what the model is learning from the inputs. For example, if we cluster training samples based on their latents, we can determine whether models are in fact generating similar representations for queries that are semantically similar.

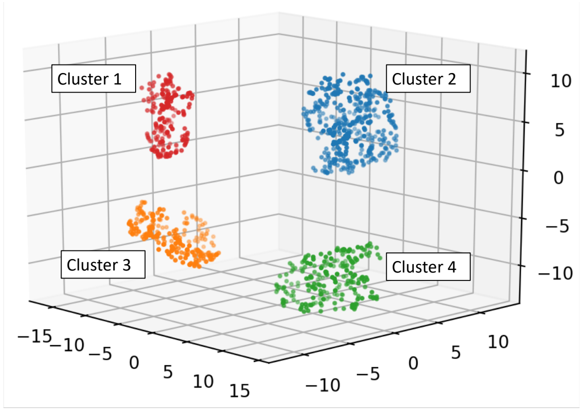


Figure 4.14: Clustering dimensionality-reduced latents for the NN (100w, 1d) model on the TPC-H dataset.

We use the t-SNE technique to cluster latents, which is a dimensionality reduction technique that is often used to visualize high-dimensional data [128]. This approach has an objective function that minimizes the KL-divergence between a distribution that measures pairwise similarities of the objects in high-dimensional space and a distribution that measures the pairwise similarities of the corresponding low-dimensional points [128]. Compared to principal component analysis (PCA), t-SNE is non-linear and relies on probabilities.

We cluster latent vectors from the (100w, 1d) NN model for the $6Join$ training set from each dataset. In Figure 4.14, we reduce the dimensionality of the latents from the (100w, 1d) model on the TPC-H dataset (100 hidden units total) down to three dimensions, which is the highest number of dimensions allowed for t-SNE. In the figure, there are four clusters, each representing different sets of joins:

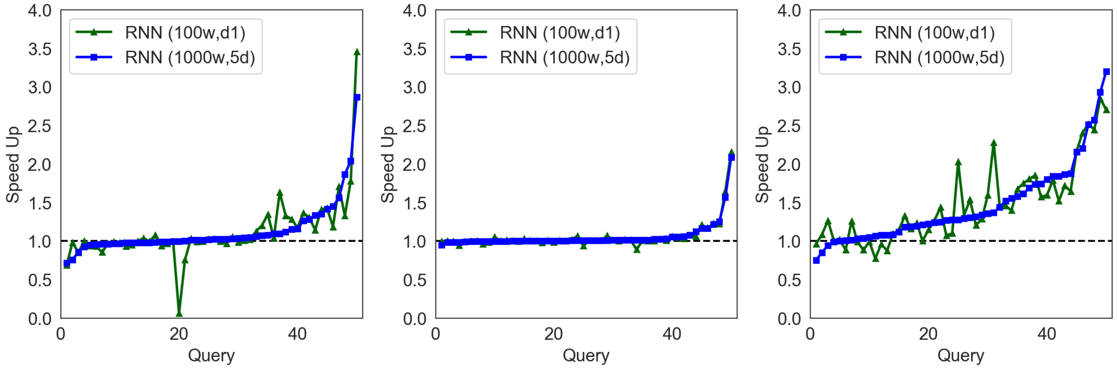
- Cluster 1: customer,lineitem,nation,orders,partsupp,region

- Cluster 2: customer,lineitem,orders,part,partsupp,supplier
- Cluster 3: customer,lineitem,nation,orders,partsupp,supplier
- Cluster 4: customer,lineitem,nation,orders,part,partsupp

For t-SNE, the distance between clusters is irrelevant, the more important factor is the relevance among the points that are clustered together. For the DMV dataset and IMDB, the clusters do not represent combinations of relations, but we observe that queries that are near each other share similar selection predicate values.

For the RNN (100w, 1d) model, we find that clusters are determined based on the sequence of operations. Recall, during training, the RNN learns to predict cardinalities for different join sequences, as a result of observing many queries. We observe that the resulting clusters represent queries that end with similar operations. For example, one cluster contains combinations of relations $orders$, $lineitem$, $partsupp$, but always ends the sequence with joins on either the $supplier$ and $part$ relation or $customer$ and $supplier$. We find that complex models (1000w, 5), also show a similar trend. This is actually a side-effect of RNNs, as more recent actions have a heavier influence on the content that exists in the hidden states. More specifically, it is difficult to learn long-term dependencies as the gradient is much smaller compared to short-term interactions [52].

As an additional experiment, we cluster the latents from queries that have not been included in the training. Ideally, although these queries have never been observed by the model, they should cluster with similar training queries. We focus on the TPC-H join removal scenario, originally shown in Figure 4.13b. When we cluster the latents from the (1000w, 5d) NN model, the queries that were not included in the training are clustered separately from the rest. This seems to imply that the NN does not learn the interactions between subqueries. This is not the case for the RNN, as queries that are left out of training are clustered together with queries that have similar subqueries. For example, a query that joins relations $lineitem$, $orders$, $partsupp$, $customer$, $part$, and $nation$, is clustered together with queries that contain relations $lineitem$, $orders$, $partsupp$, $customer$, $part$, and $supplier$.



(a) IMDB Impact

(b) DMV Impact

(c) TPC-H Impact

Figure 4.15: Query execution time speed-ups thanks to cardinality estimates from simple or complex RNN

4.4 Practical Considerations

In this section, we study two additional practical considerations. In Section 4.3, we evaluated the accuracy of cardinality estimates produced by the different models. In this section, we evaluate whether the cardinality estimate improvements lead to faster query execution plans. Additionally, in earlier sections, we showed the trade-offs between prediction error and time overhead due to model training. We did not consider the time that it takes to execute the training queries. To minimize this overhead, we consider using active learning as a way to reduce the time spent generating training sets.

4.4.1 Impact on Query Plans

We evaluate the impact of these models starting with a simple RNN model (100w, 1d) and going to a more complex one (1000w, 5d). We use the RNN, as query optimization requires evaluating cardinalities for several possible subqueries that could exist in the final plan. We evaluate the performance benefit for queries with 6 relations for each of the three datasets. As we collect the subquery cardinalities from the RNN, these estimates are then fed into a version of PostgreSQL modified to accept external cardinality estimates [24].

In Figure 4.15, we show the performance impact of these improved cardinalities compared to the default cardinality estimates from PostgreSQL. First, for the IMDB dataset, we show the performance improvement for 50 queries in Figure 4.15a. The runtimes for these queries range from <1sec up to 200sec. The simple RNN model improves the performance of 54% of the queries, while the complex model improves 60% of the queries. For the simpler model, query 22 is an outlier where the model's estimates actually slows down the query considerably (from 2 seconds up to 39 seconds). In contrast, there is no significant slow down on any query for the complex model.

For the DMV dataset, both the simple model and complex model improve the performance for 76% of the queries and there is no significant slow down for any query. We should note, however, that a majority of the query runtimes in this dataset range from 1 to 3 seconds. Finally, for the TPC-H dataset, the complex model improves 90% of the queries. The simpler model also makes a significant improvement, speeding up 84% of the queries. The query execution times for this dataset range from 20 to 120 seconds.

4.4.2 Reducing the Training

Building a model can be time consuming depending on the size of the model, the training time, and the amount of time that it takes to collect the training samples. To train the models shown in Section 4.3, we needed to run a large set of random queries to collect their ground-truth cardinalities, the output Y, for the models. Depending on the complexity of these queries, running them and collecting these labels can be time consuming. This process can be parallelized, but it comes with a resource cost.

Models can be trained in several ways. One approach to reducing the time to collect training samples, is to train the model in an online fashion. That is, as the user executes queries while using the system, the model can train on only those queries. The learning happens in an incremental fashion, and updates the model after observing a batch of samples. This approach can work well if the user executes similar queries. Online learning can also be fast and memory efficient, but the learning may experience a drift [47], where the model's decision boundary changes largely depends on the latest samples it observes.

Alternatively, instead of relying on a user to provide query samples, we can use a technique known as *active learning*. Active learning selects the best sample of candidates to improve a model's objective and to train as effectively as possible [71]. It is ideal in settings where labeled examples are expensive to obtain [26].

Active learning works through a series of iterations. In each iteration, it determines unlabeled points to add to the training sample to improve the model. Given a large pool of unlabeled samples, active learning will select the unlabeled sample that should be annotated to improve the model's predictions. In our context, given a large pool of unlabeled queries, active learning should help narrow down which queries to execute next.

There are various existing active learning methods. Common techniques include using uncertainty sampling, query-by-committee (QBC), and expected model change [71]. In this work, we focus on using QBC [117]. After each active learning iteration, QBC first builds a committee of learners from the existing training dataset via bootstrapping [137]. Each learner in the committee makes a prediction for all the samples in the unlabeled pool. The sample with the highest disagreement is labeled and added to the training pool. For regression tasks, this disagreement can be measured by the variance in the predictions across the learners [111].

Traditionally, active learning only adds a single informative sample in each data sampling iteration [25]. More recently, *batch-model* AL (BMAL), where multiple samples are labeled in each iteration has become more prevalent, as labeling in bulk or in parallel has been more accessible in recent years [27]. As shown in work by Wu et. al. [137] careful attention must be placed in picking out diverse points with BMAL, as models might disagree on a batch that contains very similar points, leading to suboptimal results.

We use BMAL in the following experiment and run three different methods to help select the unlabeled points for each iteration:

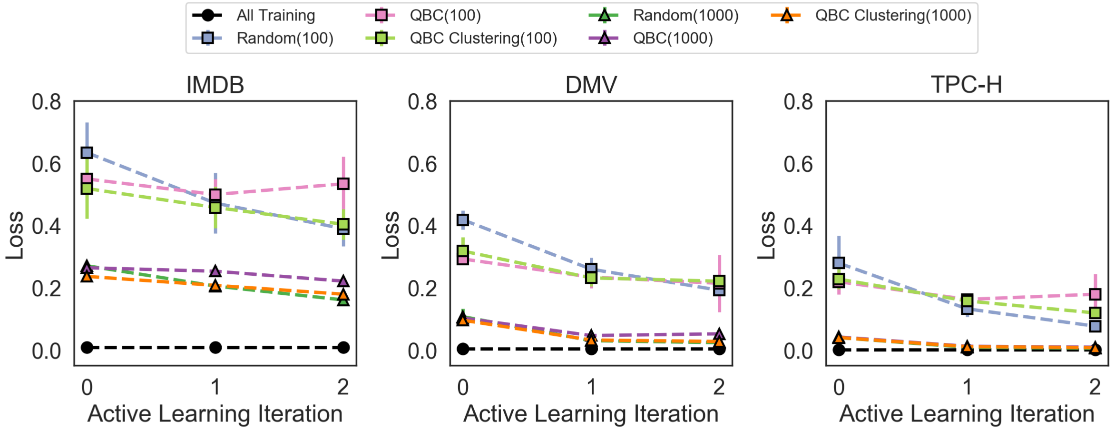


Figure 4.16: Active Learning

1. **QBC**: after each iteration, we train an ensemble of models and select the top K points with the highest disagreement

2. **QBC+Clustering**: we train an ensemble of models, but pick out the top K *diverse* set of points through clustering, which is based on the technique from [137] for linear regression

3. **Random**: we select a random sample of points from the unlabeled pool

For each dataset, we use training samples from the $2Join$, $4Join$, and $6Join$ set along with all their subqueries, for a total of 600K samples for the model. We run two experiments. In the first experiment, we start with a small number of training samples (100) and set K=100. For the second experiment, we start with a larger sample (1000) and set K=1000. As the number of training samples is small, we include regularization to prevent overfitting.

In Figure 4.16, we show the loss of each technique for three active learning iterations on each dataset. We show the results for both experiments (K=100 and K=1000). Each point represents the average loss for three separate runs. For each graph, we also include the loss for the case where all samples are labeled (labeled as “all training”).

In general, we find that with small training sets, QBC and QBC+Clustering result in a lower loss, particularly at the end of the first iteration. For subsequent iterations, the random technique performs just as well and in some cases even better, as in the TPC-H dataset for example. QBC is competitive, but it often overfits as shown by the cases where the loss increases (IMDB and TPC-H). This is expected, as BMAL techniques are known to select a distinct set of points to improve the loss more effectively.

When the training set is larger (K=1000), all techniques perform similarly, negating the immediate benefit of active learning. Nevertheless, adding fewer points rather than the entire training set can still reach a loss that is approximately an order of magnitude away from the loss that includes all the training data.

4.5 Summary

We show the promise of using deep learning models to predict query cardinalities. In our study we found that even simple models can improve the runtimes of several queries across a variety of datasets. Although there is a large training overhead, we can use techniques such as active learning to reduce the loss quickly without having to run a large set of queries.

Chapter 5

RELATED WORK

In this chapter, we cover related work for the three previous chapters. We cover the literature in the following areas: query performance prediction, admission control, elastic scaling, workload consolidation, and cardinality estimation.

5.1 Related Work on Generating Personalized Service Level Agreements

There are several lines of work related to PSLAManager from Chapter 2. The key contribution of the PSLAManager is to summarize a query workload into a concise PSLA with latency-based guarantees. Related work includes work that considers query latency prediction and SLAs for cloud data analytics. In addition, there is other work that includes various templating and summarization methods. We review these techniques here.

Query Runtime Prediction Related work has relied on various types of techniques to predict query execution times. Work by Gandhi et. al. [50] uses gray-box regression models for Hadoop workloads that relate execution times to resource allocation parameters. The Jockey system [45] provides latency SLOs for jobs in SCOPE by using a simulator that captures information about the current job and historical traces. Jockey then dynamically adjusts resources based on the different stages of the job. Work by Yadwadkar et. al. [141] uses a set of benchmarks with diverse resource requirements to profile information about each VM type. As these benchmarks are independent of the query workloads, the benchmarks only need to be run once. Users then provide representative workloads to enable the system to recommend the best type of VM to run the job. The Ernest [129] system helps users select between configurations by sampling the input data to create a performance model. The goal here is to understand how different stages of a MapReduce job behave in a long pipeline. Herodotou et. al. [60], assumes a previously profiled workload to predict the runtime throughout different sized clusters. Jalaparti et. al. [62] generates resource combinations given performance goals. In contrast to the above approaches, PSLAManager focuses on ad-hoc analytical queries with no prior profiles. The system relies on a previously built model based on a different dataset to predict the performance of the queries.

Query Admission Control Admission control frameworks provide techniques to reject queries that cannot meet SLA objectives. Q-Cop [126] considers the number of concurrent queries for each query type. Based on this query mix, it uses a linear regression model to estimate the running time of the query with the goal of minimizing wait time. In contrast, ActiveSLA [139] builds a non-linear classification model to predict the probability for each query to meet or miss its deadline. In addition, it takes as input information about each query and system-level metrics. Both of these systems use a system-wide fixed query performance threshold and reject queries that could affect the overall goal. PSLAManager, in contrast, does not reject any queries. It groups query templates in terms of estimated runtime and presents to the user different tiers with a mix of runtime, price, and query capabilities, given those groups of query templates.

Workload Summarization The idea of representing multiple queries with a single template is not new. The term "query template" is commonly used to refer to parameterized queries. Rajaraman et al. [109] used the term query template to designate parameterized queries that differ in their selection predicates. Agarwal et al. [6] use the term query template to refer to queries with different projected attributes. We generalize the notion of a query template to include queries that differ in the projected attributes, relations used in joins (the joined tables are also parameters in our templates), and selection predicates.

Perhaps, the work in the literature closest to ours is by Chaudhuri et al. [30, 29], in the context of SQL workload compression. The main goal of their work is to compress workloads of queries to improve scalability of tasks such as index selection, without compromising the quality of such tasks' results. In contrast, our goal is to cluster all possible queries by runtime estimates and combine queries into templates guaranteeing coverage in terms of query capabilities. Our optimization goal and workload compression method are thus different.

Howe [61] have looked at the problem of generating sample queries from user databases. Their work is complementary to our workload generation approach as it focuses on inferring possible joins between tables, while we assume the presence of a schema with PK-FK constraints.

5.2 Related Work on Meeting Performance Guarantees in the Cloud

Elastic Scaling for Performance Guarantees Performance guarantees have been the focus of real-time database systems [66], where the goal is to schedule queries in a fixed-size cluster and ensure they meet their deadlines. Work by Das et. al [37], uses telemetry to determine whether to scale up containers within a single node, whereas our goal is to scale the number of containers per query. Ernest [129] builds performance models based on the behavior of the job on small samples of the data. CherryPick [8] uses a similar job sampling approach, but uses Bayesian Optimization to build performance models. Morpheus [64] relies on historical usage to determine SLOs and enforce these SLOs by using scheduling techniques to isolate jobs. Several of these systems require representative workloads or repeated tenant usage patterns. In our work, performance guarantees are based on an offline model without prior knowledge about the tenant's workload. In addition, several systems have also studied performance SLAs through dynamic resource allocation, through feedback control [80] and through reinforcement learning techniques as seen in TIRAMOLA [70]. Others leverage decisions based on resource utilization goals [36, 40, 46, 91, 124, 140]. In PerfEnforce, we compare a variety of query scheduling algorithms including feedback control, reinforcement learning, and online learning techniques.

Cloud providers today offer the ability to scale a database application [11, 134]. However, they require users to manually specify scaling conditions through vendor-specific APIs. This requires expertise and imposes the risk of resource over-provisioning. Moreover, these scaling features can be costly, as they are subject to downtimes when data re-balancing is required [11]. Most academic work on elastic systems focuses on OLTP workloads [38, 122, 130] and thus develops new techniques for tenant database migration [43], data re-partitioning while maintaining consistency [89] or automated replication [130]. In these systems, the goal is to maximize aggregate system performance, while our focus is on a per-query performance guarantees.

Multi-Tenant Workload Consolidation An active area of research in multi-tenant cloud DBMS systems is *tenant packing* [44, 84, 82], or how best to co-locate tenants on a shared set of machines or even DBMS instances. In a multi-tenant setting, guaranteeing these objectives relies on the ability to pack tenants effectively without significantly impacting performance. Related work addresses this challenge by focusing on techniques that can help mitigate the result of a bad tenant packing by either finding a good initial tenant placement strategy or dynamically migrating tenants throughout their session [40, 44, 76, 82, 124]. To help minimize costs, the Kairos [36] system predicts hardware requirements for combined tenant workloads and similarly, Romano [100] focuses on consolidating workloads across different storage devices. Although PerfEnforce is a multi-tenant service, we do not focus on minimizing the effect of workload interference or finding satisfactory tenant packings. Our architecture isolates each tenant through the use of containers. Each tenant's dataset attaches to each of the containers, allowing each tenant to process the data in isolation.

Work by Wong et. al. [135] addresses the need to consolidate thousands of tenants by scheduling their queries on specific groups of resources. Their work focuses on scheduling tenants based on temporal skew and keeps the shared cluster at a static size for days, while PerfEnforce's simulation provisioning algorithm quickly and automatically provisions new machines if there is a high demand.

Provisioning In terms of provisioning, some systems rely on machine learning techniques such as the hill-climbing approach seen in Marcus et. al. [86], which allows machines to learn an optimal time to wait before they shut down. Neural networks for dynamic allocation [88] or dynamic provisioning [106] have also been used, but have distinct goals: one focuses on allocating resources with minimal use of electrical power while the other assumes predictable workloads.

PerfEnforce focuses on combining query scheduling and resource provisioning techniques to minimize costs for the service provider. Our query scheduling techniques focus on allocating resources at a per-query level, while provisioning focuses on allocating sufficient machines at a global level across many tenants. Similarly, SmartSLA [140] minimizes the total cost by performing a local analysis (at the client level) and a global analysis (among many tenants). In contrast to our work, SmartSLA focuses on allocating based on virtualized resources. In addition, their global analysis determines the amount of resources to provide for different tenant classes. Quasar [40], a system based on Paragon [39], focuses the combination between resource allocation and resource assignment to provide performance guarantees. Their resource assignment technique focuses on finding resources that will not impact the performance for other tenants, while resource allocation focuses on determining the amount of resources that should be used by a workload. Again, finding a good tenant placement strategy is not the focus of our work. In addition, we focus on algorithms that help determine when to launch or turn off machines.

5.3 Related Work on Using Deep Learning for Cardinality Estimation

Cardinality Estimation Cardinality estimation is an important component in the query optimization process, as it is the process of estimating the number of tuples produced by a subquery. Many optimizers today use histograms to estimate cardinalities. These structures can efficiently summarize the frequency distribution of one or more attributes. For single dimensions, histograms split the data using equal-sized buckets (equi-width) or buckets with equal frequencies (equi-depth). To minimize errors, statistics about each bucket are also stored including but not limited to the number of items, average value, and mode [34].

Related work in cardinality estimation has looked at more sophisticated histograms including multi-dimensional ones such as *mHist* [104] and *genHist* [55]. There have also been more adaptive multi-histogram approaches such as *stHoles* [23], which is generated by analyzing query results, compared to analyzing the entire dataset. Unfortunately, as more dimensions exist in the data, the more difficult it is to build these multi-dimensional models [42]. In addition, constructing these multi-dimensional histograms can take a long time.

There are other existing alternatives to summarize data, including wavelets, graphical models [127], sketches [34], and sampling [77]. Nevertheless across of these these techniques, there is always a trade-off across time, space and accuracy of these models.

Learning Optimizers Leo [121], was one of the first approaches to automatically adjust an optimizer's estimates based on past mistakes. This requires successive runs of similar queries to make adjustments.

Similarly, in the effort of using a self-correcting loop, others have proposed a “black-box” approach to cardinality estimation by grouping queries into syntactic families [85]. Machine learning techniques are then used to learn the cardinality distributions of these queries based on features describing the query attributes, constants, operators and aggregates. They specifically focus on applications that have fixed workloads do not require fine-grained, sub-plan estimates.

Work by Marcus [87] uses a deep reinforcement learning technique to find optimal join orders to improve query latency on a fixed database. They use cost estimates from PostgreSQL to bootstrap the learning and continuously improve the accuracy of the model’s rewards during training. Related work by Sanjay [74], also uses deep reinforcement learning to improve query plans, but they assume perfect cardinality predictions for base relations.

Neural Networks and Cardinality Estimation Liu [81] use neural networks to solve the cardinality estimation problem, but focus on selection queries only. Hasan [58] also only focus on selectivity estimation, but show that deep learning models are particularly successful at predicting query cardinalities with a large number of selection predicates.

Work by Kipf [69] proposes a new deep learning approach to cardinality estimation by using a multi-set convolutional network. Cardinality estimation does improve, but they do not show improvement of query plans. In addition, our work explores the space, time, accuracy of these models across a variety of datasets.

Work by Kraska [73] uses a mixture of neural networks to learn the distribution of an attribute with a focus on building fast indexes. In SageDB [72], this work is extended towards building a new system that learns the underlying structure of the data to provide optimal query plans. In their work, they state that one key aspect in successfully improving these query plans is through cardinality estimation. They are currently working on a hybrid model-based approach to cardinality estimation, where they balance between looking for a model that can learn the distribution of the data and a model that can capture the extreme outliers and anomalies of the data.

Wu [136] learn several models to predict the cardinalities for a variety of template subgraphs in a dataset instead of building one large model. Input features include filters and parameters for the subgraph, but they do not featurize information about the dataset (i.e. the relations). Thus, their models cannot make predictions for unobserved subgraph templates.

Chapter 6

CONCLUSION AND FUTURE DIRECTIONS

Data analytics has been a critical asset for large enterprises that need to explore and gain further insights from their data. Cloud computing has opened data analytics to the masses, providing an efficient way to acquire resources at scale compared to an on-premise cluster. Today, when configuring a data analytics service in the cloud, users are required to pay for a cluster configuration. This creates a challenge, as users are required to translate their data management needs into resources. We proposed to change this interface and instead, show users what they can do with their data along with a set price. In this dissertation, we introduced the notion of a “Personalized Service Level Agreement”, which is a performance-based SLA specific to a user database. We then developed and presented three new components: PSLAManager, SLAOrchestrator (which introduced the PerfEnforce subsystem), and DeepQuery.

We first focused on the challenge of generating performance-based SLAs for each user. To address this problem, we presented PSLAManager (Chapter 2). PSLAManager generates Personalized Service Level Agreements that show users a selection of service tiers with different price-performance options to analyze their data using a cloud service. To use this service, users first upload their data and schema to PSLAManager, allowing it to collect statistics about the data. Based on these statistics, PSLAManager generates a query workload, estimates the runtime of this workload on different configurations, compresses the queries, and provides the user with a final PSLA. Each PSLA comes with several options, each with a different price and performance trade-off. We consider the PSLA an important direction for making cloud DMBSs easier to use in a cost-effective manner.

There are several direct extensions to the initial PSLA generation approach presented in Chapter 2. First, our approach currently assumes no indexes. We posit that physical tuning should happen once the user starts to query the data. The cloud can use existing methods to recommend indexes. It can then re-compute PSLAs but, this time, include the specific queries that the user is running and combinations of indexes. This approach, however, requires an extended model for query time prediction and makes it more difficult to compare service tiers with different indexes because each index accelerates different queries.

Second, many variants of the PSLA approach are possible: We could vary the structure of the query templates and the complexity of the queries shown in the PSLAs. We could use different definitions for PSLA complexity and error metrics.

Beyond these fundamental extensions, additional questions remain: Can the cloud use the PSLAs to help users reason about the cost of different queries? Can it help users to rewrite expensive queries into cheaper, possibly somewhat different, ones? What else should change about the interface between users and the DBMS now that the latter is a cloud service?

In Chapter 3 we presented SLAOrchestrator, which combines the previous PSLAManager's SLA generation with a new sub-system called PerfEnforce. SLAOrchestrator uses a double learning loop that improves SLAs. After the completion of a tenant's hourly session, PSLAManager uses the metadata collected from previous query runtimes to improve subsequent SLAs. SLAOrchestrator also improves resource management over time through PerfEnforce. PerfEnforce includes an efficient combination of elastic query scheduling and multi-tenant resource provisioning algorithms that work toward minimizing service costs. Experiments demonstrate that SLAOrchestrator dramatically reduces service costs for a common type of per-query latency SLAs.

A challenge raised by the performance-based SLAs relates to query runtime guarantees. The initial performance predictions are based heavily on an offline model. Building and training a model that generalizes well for many tenant datasets is a challenge as tenant datasets can vary in size, column distributions and schema.

The SPJ query workloads we use throughout the evaluation allowed us to demonstrate a proof of concept for SLAOrchestrator. As future work, incorporating more complex query workloads (e.g., considering aggregates and subqueries), would impact the online learning and contextual multi-armed bandit (CMAB) techniques as they would require more advanced models and more extensive feature engineering than those considered in this work. Second, for a more thorough provisioning evaluation, running SLAOrchestrator against real tenant traces would help to better understand how the system would behave under more bursty workloads.

Finally, in Chapter 4, we presented DeepQuery, where we evaluated deep learning for cardinality estimation by studying the accuracy, space and time trade-offs across several deep learning architectures. We found that simple deep learning models can learn cardinality estimations across a variety of datasets (reducing the error by 72% - 98% on average compared to PostgreSQL). In addition, we empirically evaluated the impact of injecting cardinality estimates produced by deep learning models into the PostgreSQL optimizer. In many cases, the estimates from these models lead to better query plans across all datasets, reducing the runtimes by up to 49% on select-project-join workloads.

There are several question that remain in this work. First, we have highlighted distributions that are difficult to learn for the deep models, but it is unclear why the models fail to learn specific distributions. The next step would be to generate several different types of synthetic distributions to gain a better understanding of which types of distributions the model fails to predict properly. The second challenge, which ties closely to the first, is the lack of model interpretability. Unlike the random forest models, it is difficult to understand what the models are actually learning. Given this ambiguity, there is always the chance that the model might predict cardinalities with high errors for specific subqueries, making it difficult to debug and improve the models.

Final Remarks: Providing and enforcing performance-based service level agreements at a low cost is a challenging problem. This thesis contributes techniques that can help a cloud data analytics service enforce these guarantees in a cost-effective manner. In addition, we can strengthen these techniques by relying on improved query cost estimates through trained deep learning models. In general, cloud computing has fundamentally transformed the way users administer database systems. Users now have the capability to launch resources in a matter of seconds. As these systems mature, we should continue to consider other ways we can improve the usability of these systems while minimizing the financial burden to the service providers.

BIBLIOGRAPHY

[1] Azure Data Lake Analytics. https://azure.microsoft.com/en-us/services/data-lake-analytics/.

[2] Myria. http://demo.myria.cs.washington.edu.

[3] Statistics collection recommendations - Teradata. http://knowledge.teradata.com/KCS/id/KCS015023.

[4] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[5] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

[6] S. Agarwal et al. BlinkDB: queries with bounded errors and bounded response times on very large data. In *EuroSys*, pages 29–42, 2013.

[7] M. Akdere, U. Çetintemel, M. Riondato, E. Upfal, and S. B. Zdonik. Learning-based query performance modeling and prediction. In *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*, pages 390–401, 2012.

[8] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 469–482, Boston, MA, 2017. USENIX Association.

[9] Amazon EBS. https://aws.amazon.com/ebs/.

[10] Amazon Migrates 50 PB of Analytics Data from Oracle to AWS. https://aws.amazon.com/solutions/case-studies/amazon-migration-analytics/.

[11] Amazon AWS. http://aws.amazon.com/.

[12] Amazon Elastic Compute Cloud (Amazon EC2). http://www.amazon.com/gp/browse.html?node=201590011.

[13] Amazon Elastic MapReduce (EMR). http://aws.amazon.com/elasticmapreduce/.

[14] Amazon Cloud Relational Database Service (RDS). http://aws.amazon.com/rds/.

[15] Amazon Simple Storage Service (Amazon S3). http://www.amazon.com/gp/browse.html?node=16427261.

[16] Apache YARN. http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[17] Apache Spark: Lightnight-fast cluster computing. http://spark.apache.org/.

[18] Ashish Thusoo et. al. Hive - a petabyte scale data warehouse using Hadoop. In *Proc. of the 26th ICDE Conf.*, 2010.

[19] Azure storage. https://azure.microsoft.com/en-us/services/storage/.

[20] Azure virtual machines. https://azure.microsoft.com/en-us/services/virtual-machines/.

[21] Google BigQuery. https://developers.google.com/bigquery/.

[22] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[23] N. Bruno, S. Chaudhuri, and L. Gravano. STHoles: A multidimensional workload-aware histogram. In *SIGMOD Conference*, 2001.

[24] W. Cai, M. Balazinska, and D. Suciu. Pessimistic cardinality estimation: Tighter upper bounds for intermediate join cardinalities. In *SIGMOD*, 2019.

[25] W. Cai, M. Zhang, and Y. Zhang. Batch mode active learning for regression with expected model change. *IEEE Transactions on Neural Networks and Learning Systems*, 28(7):1668–1681, July 2017.

[26] W. Cai, Y. Zhang, and J. Zhou. Maximizing expected model change for active learning in regression. *2013 IEEE 13th International Conference on Data Mining*, pages 51–60, 2013.

[27] S. Chakraborty, V. Balasubramanian, and S. Panchanathan. Adaptive batch mode active learning. *IEEE Transactions on Neural Networks and Learning Systems*, 26(8):1747–1760, Aug 2015.

[28] O. Chapelle and L. Li. An empirical evaluation of thompson sampling. In *Advances in Neural Information Processing Systems 24*. 2011.

[29] S. Chaudhuri, P. Ganesan, and V. R. Narasayya. Primitives for workload summarization and implications for SQL. In *VLDB*, pages 730–741, 2003.

[30] S. Chaudhuri, A. K. Gupta, and V. R. Narasayya. Compressing SQL workloads. In *SIGMOD*, pages 488–499, 2002.

[31] S. Chaudhuri and V. Narasayya. Program for TPC-H data generation with skew.

[32] Y. Chi et al. SLA-tree: a framework for efficiently supporting sla-based decisions in cloud computing. In *Proc. of the EDBT Conf.*, pages 129–140, 2011.

[33] Y. Chi, H. J. Moon, and H. Hacigümüs. iCBS: Incremental costbased scheduling under piecewise linear SLAs. *PVLDB*, 4(9):563–574, 2011.

[34] G. Cormode, M. N. Garofalakis, P. J. Haas, and C. Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends in Databases*, 4(1-3):1–294, 2012.

[35] A. Criminisi and J. Shotton. *Decision Forests: A Unified Framework for Classification, Regression, Density Estimation, Manifold Learning and Semi-Supervised Learning*, volume 7, pages 81–227. NOW Publishers, January 2012.

[36] C. Curino, E. P. C. Jones, R. A. Popa, N. Malviya, E. Wu, S. Madden, H. Balakrishnan, and N. Zeldovich. Relational cloud: a database service for the cloud. In *Proc. of the Fifth CIDR Conf.*, pages 235–240, 2011.

[37] S. Das, F. Li, V. R. Narasayya, and A. C. König. Automated demand-driven resource scaling in relational database-as-a-service. In *Proceedings of the 2016 International Conference on Management of Data*, Proc. of the ACM SIGMOD International Conference on Management of Data, pages 1923–1934, New York, NY, USA, 2016. ACM.

[38] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi. Albatross: Lightweight elasticity in shared storage databases for the cloud using live data migration. *Proc. of the 37th VLDB Conf.*, 4(8):494–505, 2011.

[39] C. Delimitrou and C. Kozyrakis. Paragon: QoS-aware scheduling for heterogeneous datacenters. *SIGPLAN Not.*, 48(4):77–88, Mar. 2013.

[40] C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and QoS-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 127–144, 2014.

[41] M. Denil, D. Matheson, and N. D. Freitas. Narrowing the gap: Random forests in theory and in practice. In E. P. Xing and T. Jebara, editors, *Proceedings of the 31st International Conference on Machine Learning*, volume 32 of *Proceedings of Machine Learning Research*, pages 665–673, Bejing, China, 22–24 Jun 2014. PMLR.

[42] T. Eavis et al. Rk-hist: An R-tree based histogram for multi-dimensional selectivity estimation. In *CIKM '07*.

[43] A. J. Elmore, S. Das, D. Agrawal, and A. El Abbadi. Zephyr: live migration in shared nothing databases for elastic cloud platforms. In *Proc. of the SIGMOD Conf.*, pages 301–312, 2011.

[44] A. J. Elmore, S. Das, A. Pucher, D. Agrawal, A. El Abbadi, and X. Yan. Characterizing tenant behavior for placement and crisis mitigation in multitenant DBMSs. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 517–528, 2013.

[45] A. D. Ferguson, P. Bodík, S. Kandula, E. Boutin, and R. Fonseca. Jockey: guaranteed job latency in data parallel clusters. In *European Conference on Computer Systems, Proceedings of the Seventh EuroSys Conference 2012, EuroSys '12, Bern, Switzerland, April 10-13, 2012*, pages 99–112, 2012.

[46] M. Fu, A. Agrawal, A. Floratou, B. Graham, A. Jorgensen, M. Li, N. Lu, K. Ramasamy, S. Rao, and C. Wang. Twitter Heron: Towards extensible streaming engines. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 1165–1172, 2017.

[47] J. a. Gama, I. Žliobaitė, A. Bifet, M. Pechenizkiy, and A. Bouchachia. A survey on concept drift adaptation. *ACM Comput. Surv.*, 46(4):44:1–44:37, Mar. 2014.

[48] A. Ganapathi, Y. Chen, A. Fox, R. H. Katz, and D. A. Patterson. Statistics-driven workload modeling for the cloud. In *ICDEW*, pages 87–92, 2010.

[49] A. Ganapathi et al. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *ICDE*, pages 592–603, 2009.

[50] A. Gandhi, P. Dube, A. Kochut, L. Zhang, and S. Thota. Autoscaling for hadoop clusters. In *IC2E 2016*.

[51] Z. Gong, X. Gu, and J. Wilkes. PRESS: Predictive elastic resource scaling for cloud systems. In *6th IEEE/IFIP International Conference on Network and Service Management (CNSM 2010)*, Niagara Falls, Canada, 2010.

[52] I. Goodfellow et al. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[53] Google Cloud Storage. https://cloud.google.com/storage/.

[54] Google Compute. https://cloud.google.com/.

[55] D. Gunopulos, G. Kollios, V. J. Tsotras, and C. Domeniconi. Approximating multi-dimensional aggregate range queries over real attributes. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, pages 463–474, New York, NY, USA, 2000. ACM.

[56] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, Nov. 2009.

[57] D. Halperin, V. T. de Almeida, L. L. Choo, S. Chu, P. Koutris, D. Moritz, J. Ortiz, V. Ruamviboonsuk, J. Wang, A. Whitaker, S. Xu, M. Balazinska, B. Howe, and D. Suciu. Demonstration of the Myria big data management service. In *SIGMOD*, pages 881–884, 2014.

[58] S. Hasan, S. Thirumuruganathan, J. Augustine, N. Koudas, and D. Gautam. Multi-attribute selectivity estimation using deep learning.

[59] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. Springer New York Inc., New York, NY, USA, 2001.

[60] H. Herodotou et al. No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics. In *Proc. of the Second SoCC Conf.*, page 18, 2011.

[61] B. Howe, G. Cole, N. Khoussainova, and L. Battle. Automatic example queries for ad hoc databases. In *SIGMOD*, pages 1319–1321, 2011.

[62] V. Jalaparti, H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Bridging the tenant-provider gap in cloud services. In *SoCC*, pages 10:1–10:14, 2012.

[63] P. K. Janert. *Feedback Control for Computer Systems*. O'Reilly Media, Inc., 2013.

[64] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, I. n. Goiri, S. Krishnan, J. Kulkarni, and S. Rao. Morpheus: Towards automated slos for enterprise clusters. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 117–134, Berkeley, CA, USA, 2016. USENIX Association.

[65] M. Kahng, P. Y. Andrews, A. Kalro, and D. H. Chau. Activis: Visual exploration of industry-scale deep neural network models. *IEEE Transactions on Visualization and Computer Graphics*, 24:88–97, 2018.

[66] B. Kao et al. Advances in real-time systems. chapter An Overview of Real-time Database Systems, pages 463–486. Prentice-Hall, Inc., 1995.

[67] J. D. Kelleher, B. N. Mac, and D. Aoife. *Fundamentals for Machine Learning for Predictive Data Analytics*. MIT Press, 2015.

[68] M. Kiefer, M. Heimel, S. Breß, and V. Markl. Estimating join selectivities using bandwidth-optimized kernel density models. *PVLDB*, 10(13):2085–2096, 2017.

[69] A. Kipf, T. Kipf, B. Radke, V. Leis, P. A. Boncz, and A. Kemper. Learned cardinalities: Estimating correlated joins with deep learning. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*, 2019.

[70] I. Konstantinou et al. TIRAMOLA: elastic nosql provisioning through a cloud management platform. In *Proc. of the SIGMOD Conf.*, pages 725–728, 2012.

[71] K. Konyushkova, R. Sznitman, and P. Fua. Learning active learning from data. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 4225–4235. Curran Associates, Inc., 2017.

[72] T. Kraska, M. Alizadeh, A. Beutel, E. H. Chi, A. Kristo, G. Leclerc, S. Madden, H. Mao, and V. Nathan. Sagedb: A learned database system. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*, 2019.

[73] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 489–504, New York, NY, USA, 2018. ACM.

[74] S. Krishnan, Z. Yang, K. Goldberg, J. M. Hellerstein, and I. Stoica. Learning to optimize join queries with deep reinforcement learning.

[75] W. Lang, S. Shankar, J. M. Patel, and A. Kalhan. Towards multi-tenant performance SLOs. In *ICDE*, pages 702–713, 2012.

[76] W. Lang, S. Shankar, J. M. Patel, and A. Kalhan. Towards multi-tenant performance SLOs. *IEEE Transactions on Knowledge and Data Engineering*, 26(6):1447–1463, 2014.

[77] V. Leis et al. Cardinality estimation done right: Index-based join sampling. In *CIDR 2017*.

[78] V. Leis et al. How good are query optimizers, really? *Proc. VLDB Endow.*, 2015.

[79] A. Li, X. Yang, S. Kandula, and M. Zhang. CloudCmp: comparing public cloud providers. In *SIGCOMM*, pages 1–14, 2010.

[80] H. Lim et al. Automated control for elastic storage. In *ICAC*, pages 1–10, 2010.

[81] H. Liu, M. Xu, Z. Yu, V. Corvinelli, and C. Zuzarte. Cardinality estimation using neural networks. In *Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering*, CASCON '15, pages 53–59, Riverton, NJ, USA, 2015. IBM Corp.

[82] Z. Liu, H. Hacigümüs, H. J. Moon, Y. Chi, and W.-P. Hsiung. PMAX: Tenant placement in multitenant databases for profit maximization. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, New York, NY, USA, 2013. ACM.

[83] K. Lolos, I. Konstantinou, V. Kantere, and N. Koziris. Adaptive state space partitioning of markov decision processes for elastic resource management. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 191–194, 2017.

[84] H. A. Mahmoud, H. J. Moon, Y. Chi, H. Hacigümüs, D. Agrawal, and A. El-Abbadi. CloudOptimizer: Multi-tenancy for I/O-bound OLAP workloads. In *EDBT*, pages 77–88, 2013.

[85] T. Malik, R. C. Burns, and N. V. Chawla. A Black-Box Approach to Query Cardinality Estimation. In *CIDR 2007*, pages 56–67, 2007.

[86] R. Marcus and O. Papaemmanouil. Releasing cloud databases for the chains of performance prediction models. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*, 2017.

[87] R. Marcus and O. Papaemmanouil. Towards a hands-free query optimizer through deep learning. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*, 2019.

[88] D. Minarolli and B. Freisleben. Distributed resource allocation to virtual machines via artificial neural networks. In *22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 490–499, 2014.

[89] U. F. Minhas et al. Elastic scale-out for partition-based database systems. In *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering Workshops*, ICDEW '12, pages 281–288, Washington, DC, USA, 2012. IEEE Computer Society.

[90] E. Mize. *Data Analytics: The Ultimate Beginner's Guide to Data Analytics*. Teaching Nerds, 2017.

[91] V. R. Narasayya, S. Das, M. Syamala, S. Chaudhuri, F. Li, and H. Park. A demonstration of SQLVM: performance isolation in multi-tenant relational database-as-a-service. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 1077–1080, 2013.

[92] P. O'Neil, E. O'Neil, and X. Chen. The star schema benchmark. http://www.cs.umb.edu/~poneil/StarSchemaB.PDF.

[93] P. E. O'Neil, E. J. O'Neil, X. Chen, and S. Revilak. The star schema benchmark and augmented fact table indexing. In *Performance Evaluation and Benchmarking (TPCTC)*, pages 237–252, 2009.

[94] J. Ortiz, M. Balazinska, J. Gehrke, and S. S. Keerthi. Learning state representations for query optimization with deep reinforcement learning. In *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning*, DEEM'18, pages 4:1–4:4. ACM, 2018.

[95] J. Ortiz, V. T. de Almeida, and M. Balazinska. A vision for personalized service level agreements in the cloud. In *DanaC*, pages 21–25, 2013.

[96] J. Ortiz, V. T. de Almeida, and M. Balazinska. Changing the face of database cloud services with personalized service level agreements. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*, 2015.

[97] J. Ortiz et al. Perfenforce demonstration: Data analytics with performance guarantees. In *SIGMOD*, 2016.

[98] J. Ortiz, B. Lee, M. Balazinska, J. Gehrke, and J. L. Hellerstein. SLAOrchestrator: Reducing the cost of performance SLAs for cloud data analytics. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 547–560, Boston, MA, 2018. USENIX Association.

[99] O. Papaemmanouil. Supporting extensible performance SLAs for cloud databases. In *ICDEW*, pages 123–126, April 2012.

[100] N. Park, I. Ahmad, and D. J. Lilja. Romano: Autonomous storage management using performance prediction in multi-tenant datacenters. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 21:1–21:14, 2012.

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

[102] PerfEnforce: A Dynamic Scaling Engine for Analytics with Performance Guarantees (Tech Report). https://homes.cs.washington.edu/~jortiz16/papers/perfenforce_techreport_2017.pdf.

[103] Petabytes Scale Analytics Infrastructure @Netflix. https://www.infoq.com/presentations/netflix-big-data-infrastructure.

[104] V. Poosala and Y. E. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, pages 486–495, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[105] PostgreSQL. https://www.postgresql.org/.

[106] X. Qiu, M. Hedwig, and D. Neumann. *SLA Based Dynamic Provisioning of Cloud Resource in OLTP Systems*, pages 302–310. 2012.

[107] R. J. Quinlan. Learning with continuous classes. In *AUS-AI*, pages 343–348, Singapore, 1992.

[108] T. Rabl, M. Frank, H. M. Sergieh, and H. Kosch. A data generator for cloud-scale benchmarking. TPCTC'10, pages 41–56, Berlin, Heidelberg. Springer-Verlag.

[109] A. Rajaraman, Y. Sagiv, and J. D. Ullman. Answering queries using templates with binding patterns. In *PODS*, pages 105–112, 1995.

[110] K. Ren et al. Hadoop's adolescence: An analysis of hadoop usage in scientific workloads. *PVLDB*, 6(10):853–864, Aug. 2013.

[111] J. Repicky. Active learning in regression tasks, 2017.

[112] Y. Rubner et al. The earth mover's distance as a metric for image retrieval. *International Journal of Computer Vision*, 40(2):99–121, 2000.

[113] N. B. Ruparelia. *Fundamentals for Machine Learning for Predictive Data Analytics*. MIT Press, 2015.

[114] V. Satopaa, J. Albrecht, D. Irwin, and B. Raghavan. Finding a ”kneedle” in a haystack: Detecting knee points in system behavior. In *Proceedings of the 2011 31st International Conference on Distributed Computing Systems Workshops*, ICDCSW '11, pages 166–171, Washington, DC, USA, 2011. IEEE Computer Society.

[115] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, SIGMOD '79, 1979.

[116] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra. Grad-cam: Visual explanations from deep networks via gradient-based localization. *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 618–626, 2017.

[117] H. S. Seung, M. Opper, and H. Sompolinsky. Query by committee. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, COLT '92, pages 287–294, New York, NY, USA, 1992. ACM.

[118] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes. Cloudscale: elastic resource scaling for multi-tenant cloud systems. In *Proc. of the Second SoCC Conf.*, pages 5:1–5:14, 2011.

[119] S. Skansi. *Introduction to Deep Learning - From Logical Calculus to Artificial Intelligence*. Undergraduate Topics in Computer Science. Springer, 2018.

[120] SLA for Azure Cosmos DB. https://azure.microsoft.com/en-us/support/legal/sla/cosmos-db/v1_0/.

[121] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. Leo - db2's learning optimizer. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 19–28, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[122] M. Stonebraker and A. Weisberg. The VoltDB main memory DBMS. *IEEE Data Eng. Bull.*, 36, 2013.

[123] R. S. Sutton and A. G. Barto. Reinforcement learning I: Introduction, 2016.

[124] R. Taft, W. Lang, J. Duggan, A. J. Elmore, M. Stonebraker, and D. DeWitt. STeP: Scalable tenant placement for managing database-as-a-service deployments. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, 2016.

[125] Z. Tan and S. Babu. Tempo: Robust and self-tuning resource management in multi-tenant parallel databases. *Proc. VLDB Endow.*, 9(10):720–731, June 2016.

[126] S. Tozer, T. Brecht, and A. Aboulnaga. Q-Cop: Avoiding bad query mixes to minimize client timeouts under heavy loads. In *ICDE*, pages 397–408, 2010.

[127] K. Tzoumas, A. Deshpande, and C. S. Jensen. Efficiently adapting graphical models for selectivity estimation. *The VLDB Journal*, 22(1):3–27, Feb. 2013.

[128] L. van der Maaten and G. Hinton. Visualizing data using t-sne, 2008.

[129] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 363–378, Santa Clara, CA, 2016. USENIX Association.

[130] H. T. Vo, C. Chen, and B. C. Ooi. Towards elastic transactional cloud storage with range query support. *PVLDB*, 3(1):506–517, 2010.

[131] J. Wang, T. Baker, M. Balazinska, D. Halperin, B. Haynes, B. Howe, D. Hutchison, S. Jain, R. Maas, P. Mehta, D. Moritz, B. Myers, J. Ortiz, D. Suciu, A. Whitaker, and S. Xu. The Myria big data management and analytics system and cloud services. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research*, 2017.

[132] W. Wang et al. Database Meets Deep Learning: Challenges and Opportunities. *SIGMOD Record*, 2016.

[133] What is a public cloud? https://azure.microsoft.com/en-us/overview/what-is-a-public-cloud/.

[134] Azure. http://www.windowsazure.com/en-us/.

[135] P. Wong, Z. He, and E. Lo. Parallel analytics as a service. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 25–36, 2013.

[136] C. Wu, A. Jindal, S. Amizadeh, H. Patel, W. Le, S. Qiao, and S. Rao. Towards a learning optimizer for shared clouds. *PVLDB*, 12(3):210–222, 2018.

[137] D. Wu. Pool-based sequential active learning for regression. *IEEE transactions on neural networks and learning systems*, 2018.

[138] W. Wu et al. Sampling-based query re-optimization. SIGMOD 2016.

[139] P. Xiong et al. ActiveSLA: a profit-oriented admission control framework for database-as-a-service providers. In *Proc. of the Second SoCC Conf.*, page 15, 2011.

[140] P. Xiong et al. SmartSLA: Cost-sensitive management of virtualized resources for CPU-bound database services. In *IEEE Transactions on Parallel and Distributed Systems*, pages 1441–1451, 2015.

[141] N. J. Yadwadkar, B. Hariharan, J. E. Gonzalez, B. Smith, and R. H. Katz. Selecting the best VM across multiple public clouds: A data-driven performance modeling approach. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, pages 452–465, New York, NY, USA, 2017. ACM.

[142] M. D. Zeiler and R. Fergus. Visualizing and understanding convolutional networks. In D. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, editors, *Computer Vision – ECCV 2014*, pages 818–833, Cham, 2014. Springer International Publishing.