Autonomous Database Management at Scale:
Automated Tuning, Performance Diagnosis, and Resource Decentralization

by

Dong Young Yoon

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2019

Doctoral Committee:

Associate Professor Barzan Mozafari, Chair
Associate Professor Michael Cafarella
Assistant Professor Mosharaf Chowdhury
Associate Professor Karthik Duraisamy
Professor H. V. Jagadish

Dong Young Yoon

dyoon@umich.edu

ORCID iD: 0000-0001-6666-967X

© Dong Young Yoon 2019

ACKNOWLEDGMENTS

First of all, I want to thank my advisor, Barzan Mozafari. I am very grateful that I was able to start my graduate studies with him. I learned a great deal from working with him, both within and outside the realm of my research. Barzan is very straightforward and, at the same time, incredibly insightful. He continuously gave me valuable advice that helped me in not only writing a good research paper but also becoming a better researcher overall. Barzan always devoted a tremendous amount of time to my research, for which I am sincerely grateful. Without his dedication and guidance, I would not have been able to successfully finish my Ph.D.

I am sincerely grateful for my dissertation committee members, Karthik Duraisamy and Michael Cafarella. They provided valuable feedback on my dissertation, significantly improving its completeness. H.V. Jagadish was also very supportive, and his kind words following my oral defense were immensely encouraging. Mosharaf Chowdhury was very insightful, serving as a co-advisor for my research on RDMA and my dissertation. His advice and support were vital for completing my Ph.D. Without his contribution, I can see that it could have taken a couple more years.

I also want to thank a few colleagues of mine at the University of Michigan. Ning Niu helped me with my very first project, which I published as the first author. He was brilliant and productive, and at the time, he was only an undergraduate student. I learned a lot from working with Yongjoo Park, who was a postdoc in my lab and will soon begin as an Assistant Professor at UIUC. Our lengthy discussions, regarding research or otherwise, were always fascinating and invaluable.

Last, but not least, I deeply thank my family and my fiancée, JungSoo Kim. Many international students in the U.S. can only come here because of the enormous sacrifices made by their families. My family has been no exception, and God knows all the hardships and difficulties that they had to go through to have me study abroad. For this, I feel incredibly fortunate to have my family, and I am very grateful. JungSoo has always been there for me, even before I started my Ph.D. She was the very reason I could endure and overcome all the stress I faced during my graduate studies. For her, I am endlessly thankful.

TABLE OF CONTENTS

Acknowledgments . . . . **ii**

List of Figures . . . . **vii**

List of Tables . . . . **x**

Abstract . . . . **xi**

Chapter

1 Introduction **1**

1.1 Two Aspects of Database Autonomy 2
1.2 Autonomy from Human Supervision 2
1.2.1 Automated Tuning 4
1.2.2 Performance Diagnosis 4
1.3 Autonomy among Database Components 6
1.4 Overview and Contributions 7

2 DBSherlock: A Performance Diagnostic Tool for Transactional Databases **8**

2.1 Motivation 8
2.2 System Overview 10
2.2.1 Data Collection and Preprocessing 11
2.2.2 User Interface 12
2.2.3 System Output 13
2.2.4 Current Limitations 14
2.3 Predicate Generation Criterion 15
2.4 Algorithm 16
2.4.1 Creating a Partition Space 16
2.4.2 Partition Labeling 19
2.4.3 Partition Filtering 19
2.4.4 Filling the Gaps 20
2.4.5 Extracting Predicates from Partitions 21
2.4.6 Time Complexity 22
2.5 Incorporating Domain Knowledge 22
2.6 Incorporating Causal Models 24
2.6.1 Confidence of a Causal Model 26
2.6.2 Merging Causal Models 26
2.7 Automatic Anomaly Detection 27

2.8 Evaluation . . . 28
2.8.1 Experiment Setup . . . 28
2.8.2 Test Cases . . . 30
2.8.3 Accuracy of Single Causal Models . . . 30
2.8.4 DBSherlock Predicates versus PerfXplain . . . 31
2.8.5 Effectiveness of Merged Causal Models . . . 33
2.8.6 Effect of Incorporating Domain Knowledge . . . 34
2.8.7 Explaining Compound Situations . . . 35
2.8.8 Accuracy for other workloads . . . 36
2.8.9 Over-fitting and Merged Causal Models . . . 36
2.8.10 Rare Anomalies and Robustness Against Input Errors . . . 37
2.8.11 Different Parameters/Steps in DBSherlock . . . 38
2.8.12 DBSherlock's Accuracy with Automatic Anomaly Detection . . . 40
2.8.13 User Study . . . 41
2.9 Related Work . . . 42
2.10 Summary . . . 43

3 CliffGuard: A Principled Framework for Finding Robust Database Designs . . . 45

3.1 Motivation . . . 45
3.2 System Overview . . . 48
3.3 Problem Formulation . . . 49
3.4 Summary of CliffGuard's Algorithm . . . 53
3.4.1 The BNT Algorithm . . . 53
3.4.2 CliffGuard: Algorithm . . . 56
3.5 Expressing Robustness Guarantees . . . 59
3.6 Empirical Studies . . . 62
3.6.1 Experimental Setup . . . 63
3.6.2 Workloads Change Over Time . . . 65
3.6.3 Our Distance Metric Is Sound . . . 66
3.6.4 Quality of Robust vs. Nominal Designs . . . 67
3.6.5 Effect of Robustness Knob on Performance . . . 70
3.7 Related Work . . . 70
3.8 Summary . . . 71

4 Joins on Samples: A Hybrid Sampling Scheme with Optimal Sampling Strategy . . . 72

4.1 Motivation . . . 72
4.2 Background . . . 75
4.2.1 Sampling in Databases . . . 75
4.2.2 Quality Metrics . . . 76
4.2.3 Problem Statement . . . 76
4.2.4 Scope and Limitations . . . 77
4.3 Hardness . . . 78
4.3.1 Output Size . . . 78
4.3.2 Approximating Aggregate Queries . . . 79
4.4 Generic Sampling Scheme . . . 80

4.5 Optimal Sampling 82
4.5.1 Join Size Estimation: Count on Joins 83
4.5.2 Sum on Joins 85
4.5.3 Average on Joins 86
4.6 Multiple Queries and Filters 89
4.7 Empirical Studies 90
4.7.1 Experiment Setup 90
4.7.2 Join Approximation: Centralized Setting 91
4.7.3 Join Approximation: Decentralized 93
4.7.4 Join Approximation with Filters 94
4.7.5 Combining Samples 95
4.7.6 Stratified Sampling 96
4.7.7 Optimal Parameters for Stratified Sampling 97
4.7.8 Overhead: Centralized vs. Decentralized 98
4.8 Related Work 100
4.9 Summary 102

5 Distributed Lock Management with RDMA: Decentralization without Starvation . . 103

5.1 Motivation 103
5.2 Background on RDMA 106
5.2.1 RDMA-Enabled Networks 106
5.2.2 Distributed Lock Managers 107
5.3 Design Challenges 109
5.3.1 Lock Starvation 110
5.3.2 Fault Tolerance 110
5.3.3 Deadlocks 110
5.4 Our Algorithm 111
5.4.1 Assumptions 111
5.4.2 Design Criteria 111
5.4.3 Lamport's Bakery Algorithm 112
5.4.4 Lock Object Representation 113
5.4.5 Lock Acquisition 116
5.4.6 Handling Lock Conflicts 116
5.4.7 Handling Failures and Deadlocks 118
5.4.8 Lock Release 120
5.4.9 Resetting Counters 120
5.5 Supporting Additional Capabilities 121
5.5.1 Support for Long-Running Transactions 121
5.5.2 Lock Upgrades 122
5.6 Optimization 123
5.6.1 Exponential Random Backoff 123
5.6.2 Tuning the Timeout Threshold 123
5.7 Evaluation 124
5.7.1 Experiment Setup 125
5.7.2 Locking Performance For TPC-C 126

5.7.3 Scalability of DSLR . . . 128
5.7.4 Performance with Long-Running Reads . . . 128
5.7.5 Effectiveness of Our Multi-Slot Leasing . . . 130
5.8 Related Work . . . 131
5.9 Summary . . . 132
6 Conclusion . . . **133**
Bibliography . . . **136**

LIST OF FIGURES

Figures

1.1 Six levels of vehicle autonomy [1]. . . . 2
1.2 Identifying the root cause of a performance anomaly is non-trivial, as different causes may lead to the same performance pattern. . . . 5

2.1 The workflow in DBSherlock. . . . 11
2.2 DBSherlock's user interface. . . . 13
2.3 Example of finding a predicate over an attribute in the partition space. (Steps 3 & 4 are only applied to numeric attributes.) . . . 17
2.4 Different scenarios of filtering P_j in the partition space. . . . 20
2.5 An example of a causal model in DBSherlock. . . . 25
2.6 The margin of confidence and the average F1-measure of the correct causal model for different anomalies. . . . 31
2.7 (a) The margin of confidence for single versus merged causal models, (b) the ratio of correct explanations for merged causal models (if the top-k possible causes are shown to the user), and (c) the effect of the number of datasets (i.e., number of manual diagnoses required) on the accuracy of the casual model. . . . 32
2.8 Average precision, recall and F1-measure of predicates generated by DBSherlock and PerfXplain. . . . 33
2.9 Ratio of the correct causes and their average F1-measure for compound situations (when top-3 causes are shown to the user). . . . 35
2.10 Evaluation of a merged causal model with 10 datasets in terms of (a) confidence, compared to a merged causal model with 5 datasets, (b) margin of confidence, compared to a merged causal model with 5 datasets, and (c) accuracy if top-k causes are returned. 37
2.11 The effect of (a) the number of partitions on our algorithm's average confidence and computation time, (b) the anomaly distance multiplier on its confidence, and (c) the normalized difference threshold on its average confidence and number of generated predicates. . . . 39
2.12 The effect of the parameter κ_t to average F1-measure. . . . 40

3.1 The CliffGuard architecture. . . . 48
3.2 Design D_1 is nominally optimal at μ_0 while designs D_2 and D_3 are robust against an uncertainty of $\pm\Gamma$ and $\pm\Gamma'$ in our parameter μ_0, respectively. . . . 51
3.3 (a) A descent direction d^* is one that moves away from *all* the worst-neighbors ($\theta_{max} \geq 90°$); (b) here, due to the location of the worst-neighbors, no descent direction exists. . . . 54

3.4 Geometric interpretation of the iterations in BNT. 55
3.5 Many workloads drift over time ($15.5K$ queries, 6 months). 65
3.6 Performance decay of a window W on a design made for another window W_0 is highly correlated with their distance. 66
3.7 Average and worst-case performances of designers for Vertica, averaged over all windows, for workloads R1, S1, and S2. 67
3.8 Different degrees of robustness for Workload R1. 68
3.9 Different degrees of robustness for Workload S2. 68
3.10 Performance of different designers for DBMS-X on workload R1. 69

4.1 A toy example of joining two uniform samples (left) versus a uniform sample of the join (right). 73
4.2 OPT's improvement in terms of the estimator's variance for COUNT over six baselines with synthetic dataset. 92
4.3 OPT's improvement in terms of the estimator's variance for SUM over six baselines with Synthetic dataset. 92
4.4 OPT's improvement in terms of the estimator's variance for AVG over six baselines with Synthetic dataset. 93
4.5 OPT's improvement in terms of the estimator's variance over six baselines with Instacart, Movielens and TPC-H datasets. 93
4.6 Variances of the query estimators for OPT in the centralized and decentralized settings. 95
4.7 OPT's improvement in terms of the estimator's variance over six baselines in the presence of filters. 96
4.8 Variance of the query estimators for OPT (individual) and OPT (combined) for the $S\{normal,normal\}$ dataset. 97
4.9 Query estimator variance per group for for a group-by join aggregate using SUBS versus *SS_UF*. 97
4.10 Query estimator variance per group for for a group-by join aggregate using SUBS with different values of k_{key} and k_{tuple}. 99
4.11 Time taken to generate samples for Instacart and TPC-H in centralized vs. decentralized setting. 100
4.12 Total network and disk bandwidth used to generate samples for Instacart and TPC-H in centralized vs. decentralized setting. 100

5.1 The 64-bit representation of a lock object L used by previous RDMA protocols [83,192]. 109
5.2 A high-level overview of lock acquisition and release in DSLR. 113
5.3 DSLR's 64-bit representation of a lock object L. 113
5.4 An example of a transaction T_3 acquiring an exclusive lock with DSLR on a lock object L. 116
5.5 An example of a transaction T_3 resetting a lock object L to resolve a deadlock. 117
5.6 An example of a transaction T_3 resetting a lock object L to avoid overflow of counters. 120
5.7 An example of a transaction T_3 acquiring an update lock on a lock object L. 122
5.8 Performance comparison of different distributed lock managers under TPC-C (low and high contention). 124
5.9 Scalability of different distributed locking algorithms with increasing number of nodes. 127

5.10 Performance comparison between DSLR and queue-based (i.e., two-sided) lock managers with/without transaction reordering, under a modified TPC-C with long-running reads. 129
5.11 Throughput with fixed vs. multi-slot leasing under the modified TPC-C workload. . . 130

LIST OF TABLES

Tables

1.1 Dissertation Overview. 7

2.1 Ten types of performance anomalies used in our experiments. 29
2.2 Ratio of correct causes with & without domain knowledge. 34
2.3 DBSherlock's accuracy for TPC-C and TPC-E workloads. 36
2.4 DBSherlock's robustness against rare and imperfect inputs. 38
2.5 Contributions of the different steps of our predicate generation algorithm to the overall accuracy. 40
2.6 Ratio of the correct causes for different strategies. 41
2.7 The summary of the user study. 42

3.1 Summary of our real-world and synthetic workloads. 64

4.1 Notations. 77
4.2 Six UBS baselines, each with different p and q 91
4.3 Optimal sampling parameters (p and q) in centralized setting with sampling budget $\epsilon = 0.01$. 94
4.4 Optimal sampling parameters (p and q) for S{*uniform,uniform*} for different distributions of the filtered column C 94
4.5 Sampling parameters (p and q) of OPT using individual samples for different aggregates versus a combined sample (S{*normal,normal*} dataset). 95

5.1 List of notations and procedures used by DSLR. 117

ABSTRACT

Database administration has always been a challenging task, and is becoming even more difficult with the rise of public and private clouds. Today, many enterprises outsource their database operation to cloud service providers (CSPs) in order to reduce operating costs. CSPs, now tasked with managing an extremely large number of database instances, cannot simply rely on database administrators. In fact, humans have become a bottleneck in the scalability and profitability of cloud offerings. This has created a massive demand for building autonomous databases—systems that operate with little or zero human supervision.

While autonomous databases have gained much attention in recent years in both academia and industry, many of the existing techniques remain limited to automating parameter tuning, backup/recovery, and monitoring. Consequently, there is much to be done before realizing a fully autonomous database. This dissertation examines and offers new automation techniques for three specific areas of modern database management.

1. **Automated Tuning** – We propose a new generation of physical database designers that are *robust* against uncertainty in future workloads. Given the rising popularity of approximate databases, we also develop an optimal, hybrid sampling strategy that enables efficient join processing on offline samples, a long-standing open problem in approximate query processing.

2. **Performance Diagnosis** – We design practical tools and algorithms for assisting database administrators in quickly and reliably diagnosing performance problems in their transactional databases.

3. **Resource Decentralization** – To achieve autonomy among database components in a shared environment, we propose a highly efficient, starvation-free, and fully decentralized distributed lock manager for distributed database clusters.

CHAPTER 1

Introduction

Database management systems (DBMSs) are among the most critical software components in our world today. Many vital applications across enterprise, science, and government depend on database technology to derive insight from their data and make timely decisions. However, database management has become extremely demanding due to advances in database technology and its ever-increasing complexity.

Recently, many enterprises have shifted to hosting their database in the cloud to save on costs for its hardware and software. Cloud databases present many advantages to these enterprises over hosting their own database infrastructure: they are cheaper, more flexible, and more scalable. Users of cloud databases can expect them to work 24/7 under a service-level agreement (SLA) with cloud service providers (CSPs). For this, CSPs need to monitor, diagnose, and fine-tune hosted databases continuously. Previously, these tasks were mostly done manually by humans, database administrators (DBAs), but they are quickly becoming a bottleneck in database management as the number of databases managed per DBA increases rapidly.

To solve this problem, the notion of ***autonomous databases*** has gained much attention lately. For instance, Oracle [14] has introduced its autonomous database, defined as "a cloud database that uses machine learning to eliminate the human labor associated with database tuning, security, backups, updates, and other routine management tasks traditionally performed by database administrators (DBAs)." While the term itself is not limited to the cloud environment, it gets marketed as a cloud database because that is where it is currently most wanted by users. However, all existing databases, including Oracle's new autonomous database, are not yet fully autonomous and can automate only some functionalities, like backup and parameter tuning.

From both the database research community and commercial database vendors, there has been much previous work in a number of areas, such as physical design [35, 36, 72, 73, 74, 120, 123, 197, 225, 231] and parameter tuning [95, 106, 162, 232, 241], to make databases more autonomous. However, there are still many areas that are fundamentally difficult to automate, as they require human insight, such as performance diagnosis. In this dissertation, we focus on two different

aspects of autonomous databases: a) autonomy from human supervision, and b) autonomy among database components. Then, we explore three specific areas where modern database systems can improve in either of those two aspects: 1) automated tuning; 2) performance diagnosis; and 3) resource decentralization. In each of these areas, we study new frameworks and algorithms that achieve better autonomy than previous approaches. In the following sections, we briefly discuss the two aspects of database autonomy, and provide background information on the three specific areas of database management mentioned above.

1.1 Two Aspects of Database Autonomy

By definition, an autonomous database is one that can operate with little or (ideally) no human supervision, ultimately reducing operating costs of a database. The second aspect to an autonomous database is making its components autonomous from one another, providing fault-tolerance and scalability. In the subsequent sections, we discuss both of these two aspects in detail.

1.2 Autonomy from Human Supervision

The notion of autonomous databases is relatively new, and there is no standard metric that measures the level of autonomy from human supervision in the context of database systems. However, there is a well-known taxonomy for autonomous driving, called *six levels of vehicle autonomy* [84], as shown in Figure 1.1.

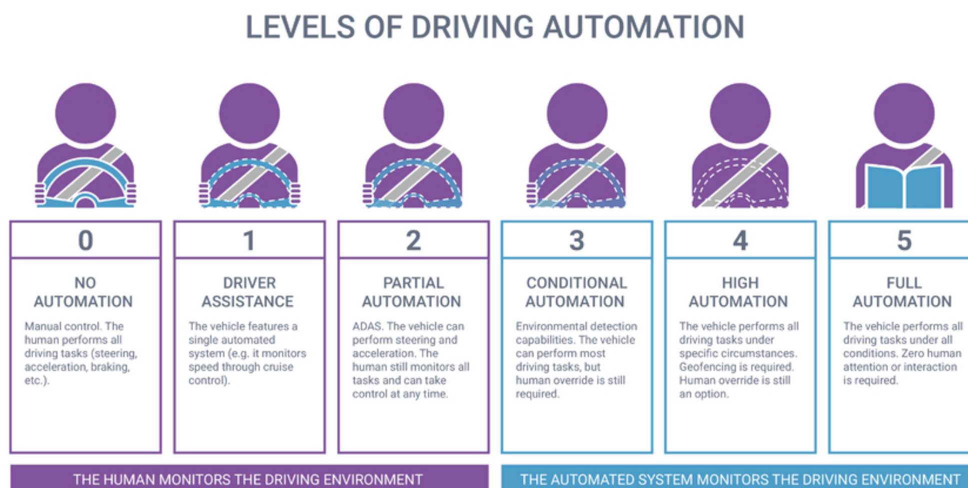


Figure 1.1: Six levels of vehicle autonomy [1].

Here, we apply a similar taxonomy to discuss levels of autonomy from human supervision in the context of database systems. Each level can be translated as follows:

- **Level 0** (Manual): A DBMS does not provide any automation. It simply provides an interface for the user and operates *manually*, merely following instructions from the user.

- **Level 1** (Assistant): A DBMS provides an assistant tool to recommend a better configuration for a DBMS's component to the user. However, it is still the user's responsibility to apply the changes and monitor the effects.

- **Level 2** (Partial): A DBMS can automatically configure or manage some components of the system, but it can still fail or conflict with the user's interests. The user needs to consistently pay attention to override the undesirable behavior from the system.

- **Level 3** (Conditional): A DBMS provides an automatic configuration or management for some components of the system in a more comprehensive way than *Level 2*, requiring less attention from the user. The user still expects to intervene from time to time.

- **Level 4** (High): A DBMS provides a complete autonomy for *some* components of the system. The user only needs to provide a high-level direction for global configuration issues.

- **Level 5** (Full): A DBMS is completely autonomous. It requires zero intervention from humans to be able to operate at full capacity.

In this dissertation, we focus on the two specific areas of database management, performance diagnosis and automated tuning, for improving the level of autonomy from human supervision. We can grade the current level of autonomy in each of the two areas as the following:

Performance Diagnosis— Most performance diagnosis tools in databases [49, 98, 105] can be classified as **Level 1**, but they are limited to looking at only database parameters and unable to explain the root cause of the performance problems. Outside of database context, decision trees [155] and robust statistics [213] have been used for automatic performance explanation of map-reduce jobs and cloud services, respectively.

Automated Tuning— Many previous auto-tuning tools for physical design [34, 35, 36, 75] and parameter tuning [95, 106, 162] can be classified as **Level 1**. Several recent works achieve **Level 3** in auto-indexing [8, 93].

Next, we provide detailed backgrounds on each of these two areas in the following sections.

1.2.1 Automated Tuning

A fundamental problem in database systems is choosing the best physical design (i.e., a small set of auxiliary structures) that enables the fastest execution of future queries. Almost all commercial databases come with designer tools [33, 34, 47, 73, 91, 221]. They create some indices or materialized views (together comprising the physical design) that they exploit during query processing. Existing designers are what we call *nominal*; that is, they assume that their input parameters are precisely known and equal to some nominal values. In practice, however, these parameters are often noisy or missing. Since nominal designers do not account for the influence of such uncertainties, they find designs that are sub-optimal and remarkably brittle. To solve this problem, a new type of database designer, called `CliffGuard` [187], is proposed, which is *robust* against parameter uncertainties, so that overall performance degrades more gracefully when future workloads deviate from the past.

Huge datasets in the cloud make it considerably difficult and expensive to achieve interactive-speed data analytics. Approximate query processing (AQP) is a viable solution to this problem, as it significantly speeds up data analytics at the cost of slightly inaccurate answers. In the context of AQP, the problem of choosing the best physical design in traditional databases translates to the problem of generating an optimal set of samples that can answer given queries with the best accuracy. However, to the best of our knowledge, there are no previous studies regarding the best sampling strategy for joining random samples. Therefore, a hybrid sampling scheme, called *Stratified-Universe-Bernoulli Sampling* (SUBS), is formalized, and its optimal sampling parameters are analyzed to improve the understanding of sampling-based joins [137].

1.2.2 Performance Diagnosis

As businesses rely on databases to support their mission-critical and real-time applications, poor database performance directly impacts their revenue and user experience. As a result, DBAs constantly monitor, diagnose, and rectify any performance decays.

Unfortunately, the manual process of debugging and diagnosing database performance problems is exceptionally tedious and non-trivial. Rather than being caused by a single slow query, performance problems in transactional databases are often due to a large number of concurrent and competing transactions. They add up to compounded, non-linear effects that are difficult to isolate. Sudden changes in request volume, transactional patterns, network traffic, or data distribution can cause previously abundant resources to become scarce, and the performance to plummet.

As an example, consider Figure 1.2, which is a graph of average transaction latencies over time. In practice, finding the root cause of this latency spike can be quite challenging. We observe nearly the same performance plot in the following situations: (i) if the overall workload suddenly

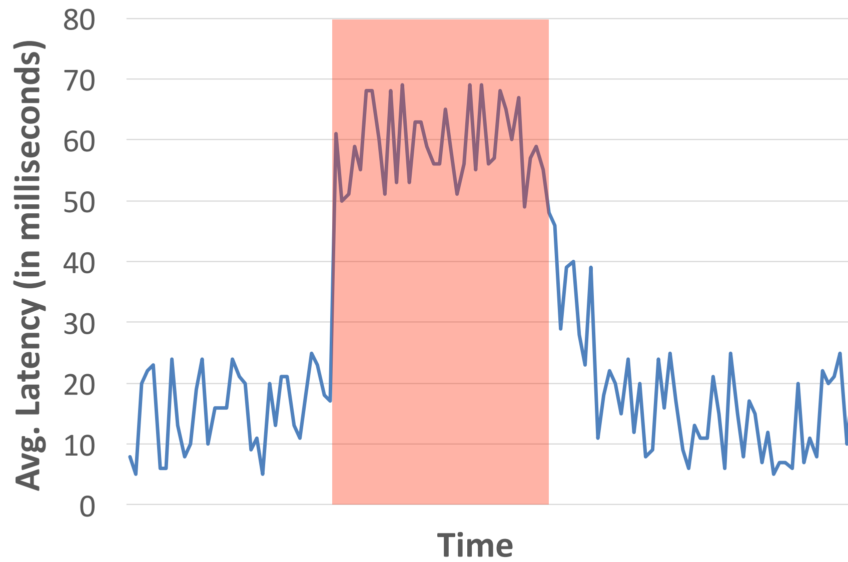


Figure 1.2: Identifying the root cause of a performance anomaly is non-trivial, as different causes may lead to the same performance pattern.

spikes, (ii) if the number of poorly written queries spikes, or (iii) if a network hiccup occurs. To establish the correct cause, the DBA has to plot several other performance metrics during the same timespan as the latency spike. The DBA has to sort through hundreds of DBMS, OS, and network telemetry, and even inspect several queries manually.

We propose a performance explanation framework called `DBSherlock` that combines techniques from outlier detection and causality analysis. The framework assists DBAs in diagnosing performance problems more quickly, more accurately, and in a more principled manner.

In the particular example of Figure 1.2, `DBSherlock`'s statistical analysis will lead to different predicates depending on the cause. When (i) has occurred, `DBSherlock` generates a predicate showing an increase in the number of lock waits and running DBMS threads compared to normal. In the case of (ii), `DBSherlock`'s predicates indicate a sudden rise of next-row-read-requests, as well as the CPU usage of the DBMS. Finally, (iii) leads to a predicate showing a lower than usual number of network packets sent or received during a specific time. In other words, `DBSherlock`'s predicates help to determine the root cause by automatically identifying appropriate signals and metrics.

1.3 Autonomy among Database Components

Autonomy among database components is typically achievable via resource decentralization. This is because, when a system has centralized resource management, every component of the system is dependent on that central resource. Jim Gray [122] once emphasized the importance of decentralization in distributed systems. His work highlighted that autonomy from decentralization makes it easy to add or remove individual nodes in a distributed system, and ultimately promotes high scalability. Similarly, the concept of an autonomous decentralized system (ADS) [184] is proposed to build a system whose components are designed to operate independently. This paradigm enables the overall system to function in the event of individual component failures, which provides fault-tolerance. Both fault-tolerance and scalability are essential features in modern distributed database systems.

Resource Decentralization— In today's database technology, resource management is centralized in most parts. Typical examples are compute, storage and lock management. For compute and storage management, resource centralization is usually handled on an application-level (e.g., SQL-on-Hadoop systems [2,3]), where applications are written as much as possible to be embarrassingly parallel so that they can use resources in a decentralized manner. For lock management, it is usually at a granularity of records, which means it is much harder for application developers to decentralize it. Therefore, there is much higher demand for decentralizing lock management than any other resource management.

In this dissertation, we focus on lock managers for achieving resource decentralization in distributed databases with RDMA(Remote Direct Memory Access)-enabled networks. This is because lock managers are a crucial component of modern distributed databases, as they dictate how resources are accessed in the system. There are centralized lock managers that can ensure fairness and prevent starvation using global knowledge of the system, but they are a single point of contention and failure. It hurts the autonomy of each node in a distributed database significantly. Also, they fall short in leveraging the full potential of RDMA networks.

On the other hand, decentralized lock managers enable each node to operate more autonomously. However, existing decentralized RDMA-based lock managers either completely sacrifice global knowledge to achieve higher throughput at the risk of starvation, or they resort to costly communications in order to maintain global knowledge, which can result in significantly lower throughput. We present DSLR, an RDMA-based decentralized lock manager that is starvation-free and promotes autonomy in distributed systems for better fault-tolerance and scalability.

Area	Framework/Algorithm	Chapter
Performance Diagnosis	`DBSherlock`: a performance explanation framework for automating performance diagnostics.	2
Automated Tuning	`CliffGuard`: a robust physical designer for databases.	3
	Joins on Samples: A hybrid sampling scheme (SUBS) with an optimal strategy derived from a theoretical analysis that achieves an information-theoretical lower bound on the lowest variance achievable by a constant factor.	4
Resource Decentralization	`DSLR`: a decentralized lock manager for RDMA-enabled networks, enhancing the autonomy of individual database instances in distributed databases.	5

Table 1.1: Dissertation Overview.

1.4 Overview and Contributions

This dissertation is organized as follows. First, we present a performance explanation framework, `DBSherlock`, that can automate significant portions of the database performance diagnostic process by combining techniques from outlier detection and causality analysis. Second, we introduce and summarize two novel approaches in automated tuning: 1) `CliffGuard` for robust physical designs; and 2) SUBS with its optimal parameters for sampling-based joins. We empirically study both approaches with an extensive set of experiments to test their effectiveness in various use-cases, and compare them against existing algorithms. Finally, we present a decentralized lock manager for RDMA-enabled networks, `DSLR`. It improves the autonomy of individual nodes in distributed databases by being fully decentralized, yet maintains high throughput and lower tail latencies, which existing decentralized lock managers fail to achieve. Table 1.1 summarizes our approaches.

The contributions in this dissertation have been presented at Computer Science conferences, including the 2015 ACM SIGMOD conference [188], the 2016 ACM SIGMOD conference [248], and the 2018 ACM SIGMOD conference [246].

CHAPTER 2

DBSherlock: A Performance Diagnostic Tool for Transactional Databases

2.1 Motivation

Many enterprise applications rely on executing transactions against their database backend to store, query, and update data. As a result, databases running online transaction processing (OLTP) workloads are some of the most mission-critical software components for enterprises. Any service interruptions or performance hiccups in these databases often lead directly to revenue loss.

Thus, a major responsibility of database administrators (DBAs) in large organizations is to constantly monitor their OLTP workload for any performance failures or slowdowns, and to take appropriate actions promptly to restore performance. However, diagnosing the root cause of a performance problem is generally tedious, as it requires the DBA to consider many possibilities by manually inspecting queries and various log files over time. These challenges are exacerbated in OLTP workloads because performance problems cannot be traced back to a few demanding queries or their poor execution plans, as is often the case in analytical workloads. In fact, most transactions take only a fraction of a millisecond to complete. However, tens of thousands of concurrent transactions competing for the same resources (e.g., CPU, disk I/O, memory) can create highly non-linear and counter-intuitive effects on database performance. Minor changes in an OLTP workload can push the system into a new performance regime, quickly making previously abundant resources scarce.

However, it can be quite challenging for most DBAs to explain (or even investigate) such phenomena. Modern databases and operating systems collect massive volumes of detailed statistics and log files over time, creating an exponential number of subsets of DBMS variables and statistics that may explain a performance decay. For instance, MySQL maintains over 260 different statistics and variables (see Section 2.2.1) and commercial DBMSs collect thousands of granular statistics (e.g., Teradata [60]). Unfortunately, existing databases fail to provide DBAs with effective tools for

analyzing performance problems using these rich datasets, aside from basic visualization and monitoring mechanisms. As a consequence, highly-skilled and highly-paid DBAs (a scarce resource themselves) spend many hours diagnosing performance problems through different conjectures and manually inspecting various queries and log files, until the root cause is found [61].

To avoid this tedious, error-prone, and adhoc procedure, we propose a performance explanation framework called `DBSherlock` that combines techniques from outlier detection and causality analysis to assist DBAs in diagnosing performance problems more easily, more accurately, and in a more principled manner. Through `DBSherlock`'s visual interface, the user (e.g., a DBA) specifies certain instances of past performance that s/he deems abnormal (and optionally, normal). `DBSherlock` then automatically analyzes large volumes of past statistics to find the most likely causes of the user-perceived anomaly, presenting them to the user along with a confidence value, either in the form of (i) concise predicates describing the combination of system configurations or workload characteristics causing the performance anomaly, or (ii) high-level diagnoses based on the existing causal models in the system. The DBA can then identify the actual cause within these few possibilities. Once the actual cause is confirmed by the DBA, his/her feedback is integrated back into `DBSherlock` to improve its causal models and future diagnoses.

Note that designing a tool for performance diagnosis is a challenging task due to the exponential number of combinations of variables and statistics that may explain the cause of a performance decay, making a naïve enumeration algorithm infeasible. Though off-the-shelf algorithms for feature selection exist, they are primarily designed to maximize a machine learning algorithm's predictive power rather than its explanatory and diagnostic power. Similarly, decision trees (e.g., PerfXplain [155]) and robust statistics (e.g., PerfAugur [213]) have been used for automatic performance explanation of map-reduce jobs and cloud services, respectively. However, such models are more likely to find secondary symptoms when the root cause of the anomaly is outside the database and not directly captured by the collected statistics. (In Section 2.6, we show that constructing and using causal models leads to significantly more relevant explanations.) Finally, while sensitivity-analysis-based techniques (e.g., Scorpion [242]) are highly effective in finding the individual tuples most responsible for extreme aggregate values in scientific computations, they are not applicable to performance diagnosis of OLTP workloads. This is because databases often avoid prohibitive logging overheads by maintaining aggregate statistics rather than detailed statistics for individual transactions. For instance, instead of recording each transaction's wait time for locks, MySQL and Postgres only record the total wait time for locks across all transactions. Thus, listing individual transactions is impractical (and often unhelpful for diagnosis, due to the complex interactions among concurrent transactions).

In this chapter, we make the following contributions:

1. To the best of our knowledge, we propose the first algorithm specifically designed to explain

and diagnose performance anomalies in highly-concurrent and complex OLTP workloads.

2. We adopt the notion of causality from the artificial intelligence (AI) literature, and treat user feedback as causal models in our diagnostic tool. We formally define a notion of *confidence* to combine predicate-based explanations and causal model predictions in a principled manner.

3. We propose a new automatic anomaly detection algorithm with a competitive accuracy to that of an expert. We also introduce a framework to prune secondary symptoms using basic forms of domain knowledge.

4. We evaluate `DBSherlock` against the state-of-the-art performance explanation technique, across a wide range of performance problems. `DBSherlock`'s predicates on average achieve 28% (and up to 55%) higher F1-measures[1] than those generated by previous techniques.

Section 2.2 describes the high-level overview of `DBSherlock`. Sections 2.3 and 2.4 present our criterion and algorithm for generating predicate-based explanations, respectively. Section 2.5 describes our technique for incorporating domain knowledge and pruning secondary symptoms from our explanations. Section 2.6 explains how our system incorporates user feedback (when available) in the form of casual models in order to provide higher-level, descriptive explanations. Section 2.7 explains how automatic anomaly detection techniques can be combined with `DBSherlock`. Section 2.8 describes our experimental results.

2.2 System Overview

`DBSherlock`'s workflow for performance explanation and diagnosis consists of six steps, as shown in Figure 2.1.

1. **Data Collection.** `DBSherlock` collects various log files, configurations, and statistics from the DBMS and OS.

2. **Preprocessing.** The collected logs are summarized and aligned by their timestamps at fixed time intervals (e.g., every second).

3. **Visualization.** Through `DBSherlock`'s graphical user interface, our end user (e.g., a DBA) can generate scatter plots of various performance statistics of the DBMS over time.

4. **Anomaly Detection.** If the end user deems any of the performance metrics of the DBMS unexpected, abnormal, or suspicious in any period of time, s/he can simply select that region of the

[1] F1-measure (a.k.a. balanced F-score) is a commonly used measure of a test's accuracy. It considers both the precision p and the recall r of the test, and is defined as: $F_1 = 2 \cdot \frac{p \cdot r}{p+r}$.

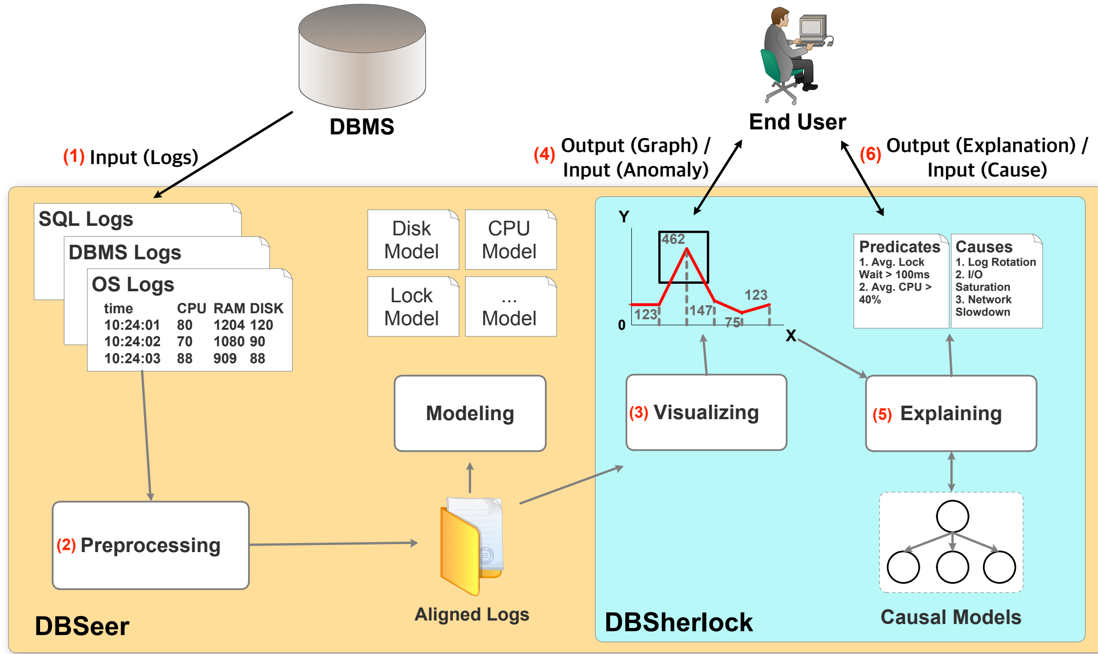


Figure 2.1: The workflow in DBSherlock.

plot and ask DBSherlock for an explanation of the observed anomaly. Alternatively, users can also rely on DBSherlock's automatic anomaly detection feature.

5. **Anomaly Explanation.** Given the user-perceived region of anomaly, DBSherlock analyzes the collected statistics and configurations over time and explains the anomaly using either descriptive predicates or actual causes.

6. **Anomaly Diagnosis and User Feedback.** Using DBSherlock's explanations as diagnostic clues, the DBA attempts to identify the root cause of the observed performance problem. Once s/he has diagnosed the actual cause, s/he provides evaluative feedback to DBSherlock. This feedback is then incorporated in DBSherlock as a causal model and used for improving future explanations.

Next, we discuss these steps in more detail: steps 1–2 in Section 2.2.1, steps 3–4 in Section 2.2.2, and steps 5–6 in Section 2.2.3. Then, we list the current limitations of DBSherlock in Section 2.2.4.

2.2.1 Data Collection and Preprocessing

We have implemented DBSherlock as a module for DBSeer [4]. DBSeer is an open-source suite of database administration tools for monitoring and predicting database performance [185, 186, 247]. We have integrated our DBSherlock into DBSeer for two reasons. First, adding performance diagnosis and explanation features will greatly benefit DBSeer's current users in gaining deeper

insight into their workloads. Second, DBSherlock can simply rely on DBSeer's API for collecting and visualizing various performance statistics from MySQL and Linux (the systems used in our experiments). Here, we briefly describe the data collection and preprocessing steps performed by DBSeer and used by DBSherlock, i.e., components (1) and (2) in Figure 2.1.

DBSeer collects various types of performance data by passively observing the DBMS and OS *in situ* (i.e., as they are running in operation), via their standard logging features. Specifically, DBSeer collects the following data at one-second intervals [185]:

(i) Resource consumption statistics from the OS (in our case, *Linux*'s /proc data), e.g., per-core CPU usage, number of disk I/Os, number of network packets, number of page faults, number of allocated/free pages, and number of context switches.

(ii) Workload statistics from the DBMS (in our case, *MySQL*'s global status variables), e.g., number of logical reads, number of SELECT, UPDATE, DELETE, and INSERT commands executed, number of flushed and dirty pages, and the total lock wait-time.[2]

(iii) Timestamped query logs, containing start-time, duration, and the SQL statements executed by the system, as well as the query plans used for each query.

(iv) Configuration parameters from the OS and the DBMS, e.g., environment variables, kernel parameters, database server configurations, network settings, and (relevant) driver versions.

DBSeer further processes this data. First, it computes aggregate statistics about transactions executed during each time interval (e.g., their average and quantile latencies, total transaction counts, etc.).[3] These transaction aggregates are then aligned with the OS and DBMS statistics and configurations according to their timestamps, using the following format:

$$(\texttt{Timestamp}, \texttt{Attr}_1, \ldots, \texttt{Attr}_k)$$

where Timestamp marks the starting time of the 1-second interval during which these data were collected, and $\{\texttt{Attr}_1, \ldots, \texttt{Attr}_k\}$ are the attributes, comprised of the transaction aggregates and other categorical and numerical metrics collected from the database and operating system. DBSherlock uses these timestamped data for its performance explanation and diagnosis purposes.

2.2.2 User Interface

DBSherlock comes with a graphical user interface, where users can plot a graph of various performance metrics over their time window of interest. This is shown as component (3) in Figure 2.1.

[2] To avoid performance overheads, DBSeer does not collect expensive statistics that are not maintained by default, e.g., fine-grained locking information.

[3] Since the number of transactions per second varies, we do not use individual query plans as attributes. Rather, we use their aggregate statistics, e.g., average cost estimates, number of index lookups.

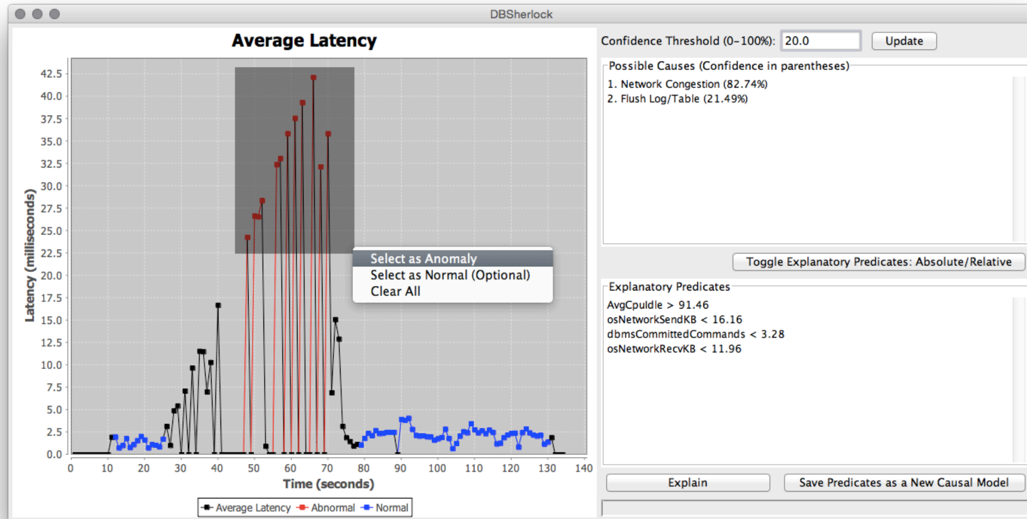


Figure 2.2: `DBSherlock`'s user interface.

For example, users might plot the average or 99% latency of transactions, number of disk I/Os, or CPU usage over the past hour, day or week. Figure 2.2 is an example of a scatter plot of the average latency of transactions over time. After inspecting this plot, the user can select some region(s) of the the graph where s/he finds some database metrics abnormal, suspicious, counter-intuitive, or simply worthy of an explanation. Regardless of the user's particular reason, we simply call the selected region(s) an *anomaly* (or call them *abnormal* regions). Optionally, the user can also select other areas of the graph that s/he thinks are normal (otherwise, the rest of the graph is implicitly treated as normal). After specifying the regions, the user asks `DBSherlock` to find likely causes or descriptive characteristics that best explain the observed *anomaly*.

When users cannot manually specify or detect the anomaly, `DBSherlock` relies on automatic anomaly detection (see Section 2.7).

2.2.3 System Output

Given a user-perceived anomaly, `DBSherlock` provides explanations in one of the following forms:

(i) predicates over different attributes of the input data; or

(ii) likely causes (and their corresponding confidence) based on existing causal models.

First, `DBSherlock` generates a number of predicates that identify anomalous values of some of the attributes that best explain the anomaly (Sections 2.3 and 2.4). For human readability,

DBSherlock returns a conjunct of simple predicates to the user.[4] For example, DBSherlock may explain an anomaly caused by a network slowdown by generating the following predicates:

$$\texttt{network_send} < \texttt{10KB} \;\wedge\; \texttt{network_recv} < \texttt{10KB} \;\wedge\; \texttt{client_wait_times} > \texttt{100ms} \;\wedge\; \texttt{cpu_usage} < \texttt{5}$$

showing that there are active clients waiting without much CPU activity. (In Section 2.6, we show that with causal models, DBSherlock can provide even more descriptive diagnoses.) Once the user identifies the actual problem (network congestion, in this example) using these predicates as diagnostic hints, s/he can provide feedback to DBSherlock by accepting these predicates and labeling them with the actual *cause* found. This 'cause' and its corresponding predicates comprise a causal model, which will be utilized by DBSherlock for future diagnoses.

When there are any causal models in the system (i.e., from accepted and labeled predicates during previous sessions), DBSherlock calculates the *confidence* of every existing causal model for the given anomaly. This *confidence* measures a causal model's fitness for the given situation. DBSherlock then presents all existing causes in their decreasing order of confidence (as long as greater than a minimum threshold). When none of the causal models yield a sufficiently large confidence, DBSherlock does not show any causes and only shows the generated predicates to the user.

Note that DBSherlock's output in either case is only a *possible* explanation/cause of the anomaly, and it is ultimately the end user's responsibility to diagnose the *actual* root cause. The objective of DBSherlock is to provide the user with informative clues to *facilitate* fast and accurate diagnosis. In the rest of this chapter, we use the terms *possible explanation* and *explanation* interchangeably, but always make a clear distinction between *possible* and *actual causes* as they are quite different from a causality perspective.

2.2.4 Current Limitations

The current implementation of DBSherlock has two limitations:

(i) DBSherlock finds an explanation for an anomaly if the anomaly affects at least one of the statistics available to the system.

(ii) Invariant characteristics of the system (e.g., fixed parameters or hardware specifications of the database server) are *not* considered a valid explanation of an anomaly.

It is straightforward to see the reason behind (i): if the anomaly does not manifest itself in any of the gathered statistics, DBSherlock has no means of distinguishing between abnormal and

[4] More complex predicates (e.g., with disjunction or negation) can easily overwhelm an average user, defeating DBSherlock's goal of being an effective tool for practitioners.

normal regions. Similarly for (ii), since the invariants of the system remain unchanged across the abnormal and normal regions, they cannot be used to distinguish the two. However, such invariants may have ultimately contributed to the anomaly in question. For instance, with a small buffer pool, dirty pages are flushed to disk frequently. Thus, when the number of concurrent transactions spikes, the pages may be flushed even more frequently. The increase in disk IOs may then affect transaction latencies. In such a case, `DBSherlock` reports the workload spike as the explanation for the increased latencies. While one might argue that small memory was the root cause of the problem, `DBSherlock` does not treat the memory size as a cause, as it is unchanged before and after the anomaly. Here, the workload spike can distinguish the two regions (see Section 2.3) and is hence returned as a cause to the user (i.e., with the justification that the memory was sufficient for the normal workload).

However, `DBSherlock`'s reported cause can still be quite helpful even in the cases above. For example, even when presented with workload spike as an explanation of the performance slowdown, an experienced DBA may still rectify the problem by modifying system invariants (e.g., provisioning a larger memory or faster disk) or throttling the additional load.

2.3 Predicate Generation Criterion

Given an abnormal region A, a normal region N, and input data T, we aim to generate a conjunct of predicates, where each predicate Pred is in one of the following forms: $\texttt{Attr}_i < \texttt{x}$, $\texttt{Attr}_i > \texttt{x}$, or $\texttt{x} < \texttt{Attr}_i < \texttt{y}$ when $\texttt{Attr}_i$ is numeric, and $\texttt{Attr}_i \in \{\texttt{x}_1, \ldots, \texttt{x}_c\}$ when $\texttt{Attr}_i$ is categorical. Intuitively, we desire a predicate that segregates the input tuples in A well from those in N. We formally define this quality as the `separation power` of a predicate.

Separation power of a predicate. Let T_N and T_A be the input tuples in the normal and abnormal regions, respectively. Also, let $\texttt{Pred}(T)$ be the input tuples satisfying predicate Pred. Then the `separation power (SP)` of a predicate Pred is defined as:

$$\texttt{SP(Pred)} = \frac{|\texttt{Pred}(T_A)|}{|T_A|} - \frac{|\texttt{Pred}(T_N)|}{|T_N|} \tag{2.1}$$

In other words, a predicate's `separation power` is the ratio of the tuples in the abnormal region satisfying the predicate, subtracted by the ratio of the tuples in the normal region satisfying the predicate. A predicate with higher separation power is more capable of distinguishing (i.e., separating) the input tuples in the abnormal region from those in the normal one. Thus, `DBSherlock`'s goal is to filter out individual attributes with low separation power.[5]

[5] This strategy is similar to *single-variable classifiers* in machine learning literature, whereby variables' individual predictive power is used for feature selection [124].

Identifying predicates with high separation power is challenging. First, one cannot find a predicate of high separation power by simply comparing the values of an attribute in the raw dataset.This is because real-world datasets and OS logs are noisy and attribute values often fluctuate regardless of the anomaly. Second, due to human error, users may not specify the boundaries of the abnormal regions with perfect precision. The user may also overlook smaller areas of anomaly, misleading `DBSherlock` to treat them as normal regions. These sources of error compound the problem of noisy datasets. Third, one cannot easily conclude that predicates with high separation power are the actual cause of an anomaly. They may simply be correlated with, or be symptoms themselves of the anomaly, and hence, lead to incorrect diagnoses. The following section describes our algorithm for efficiently finding predicates of highest separation power, while accounting for the first two sources of error. We deal with the third type of error in Section 2.5.

2.4 Algorithm

Our algorithm takes the aligned tuples as input (described in Section 2.2.1), which are separated between the abnormal and the normal regions (other tuples are ignored by `DBSherlock`).

Figure 2.3 illustrates a high-level overview of our predicate generation algorithm. The majority of our attributes are numeric (i.e., statistics), which are significantly noisier than our categorical attributes. As a result, our algorithm uses two additional steps for numeric attributes (Steps 3 and 4). (In our discussion, we highlight the differences for categorical attributes when applicable.) The first step is to discretize the domain of each attribute into a number of partitions (Step 1). Based on the user-specified abnormal and normal regions, `DBSherlock` labels each partition of an attribute as `Abnormal`, `Normal`, or `Empty` (Step 2 in Figure 2.3). Next, for numeric attributes, `DBSherlock` filters out some of the `Abnormal` and `Normal` partitions, which are mingled at this point, to find a predicate with high separation power (Step 3 in Figure 2.3). If the previous step is successful, the algorithm then fills the gap between the two separated sets of partitions and generates the candidate predicate accordingly (Steps 4 and 5 in Figure 2.3). The formal pseudo code of these steps is presented in Algorithm 1. In the rest of this section, we explain each of these steps in detail.

2.4.1 Creating a Partition Space

`DBSherlock` starts by creating a discretized domain for each attribute, called a *partition space*.

For each numeric attribute $\mathtt{Attr}_i$, we create R equi-width partitions $\mathsf{P} = \{\mathsf{P}_1, \ldots, \mathsf{P}_R\}$ that range

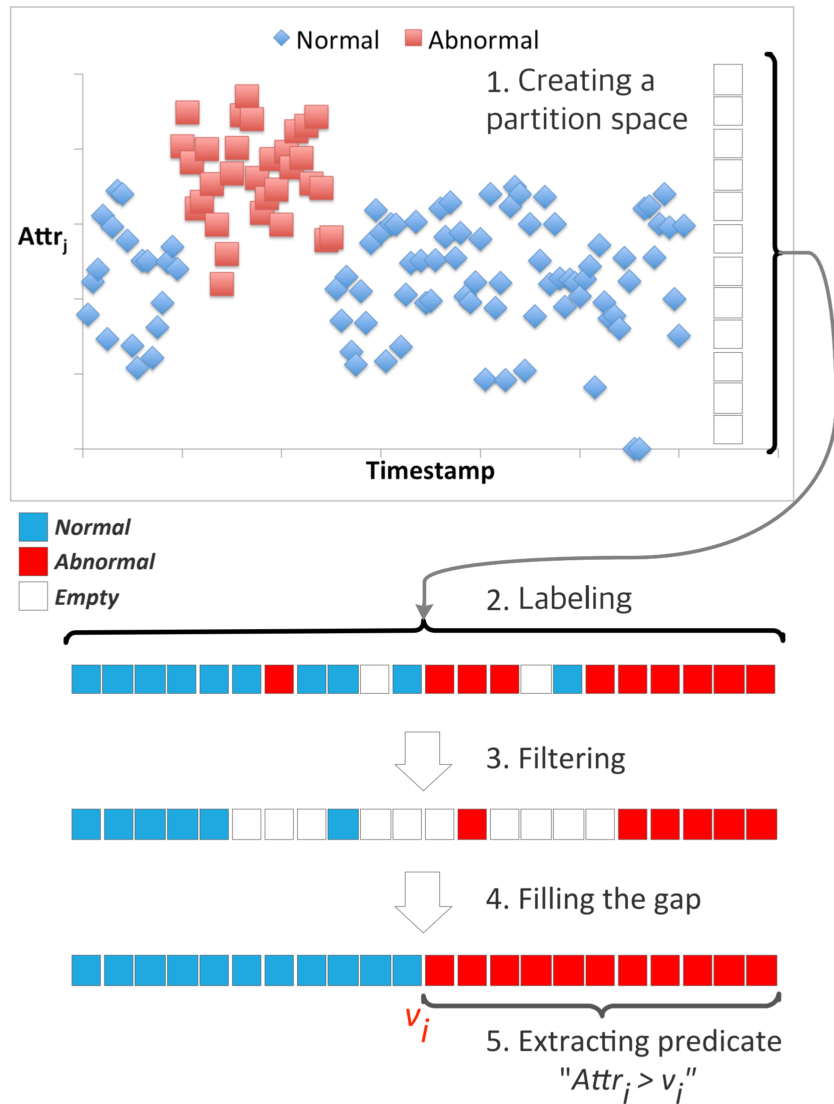


Figure 2.3: Example of finding a predicate over an attribute in the partition space. (Steps 3 & 4 are only applied to numeric attributes.)

from $\texttt{Min(Attr}_i\texttt{)}$ to $\texttt{Max(Attr}_i\texttt{)}$. The width of each partition is

$$\frac{\texttt{Max}(\texttt{Attr}_i) - \texttt{Min}(\texttt{Attr}_i)}{R}$$

We denote the lower and upper bounds of partition $\texttt{P}_j$ as $\texttt{lb(P}_j\texttt{)}$ and $\texttt{ub(P}_j\texttt{)}$, respectively. $\texttt{P}_j$ contains any value $\texttt{val}$ of $\texttt{Attr}_i$ where $\texttt{lb(P}_j\texttt{)} \leq \texttt{val} < \texttt{ub(P}_j\texttt{)}$.

For example, if the values of an attribute range from 0 to 100 and we discretize them into buckets of size 20, their partition space will be {[0,20), [20,40), [40,60), [60,80), [80,100)}. We use equi-width partitions to preserve and map the distribution of input tuples in the abnormal

Inputs: T: input tuples (with k attributes)
A: abnormal region
N: normal region
R: number of partitions
θ: normalized difference threshold
δ: anomaly distance multiplier
Output: π: list of predicates with high separation power

$\pi \leftarrow \emptyset$ **// start with no predicate**
foreach $\texttt{Attr}_i \in \{\texttt{Attr}_1, \ldots, \texttt{Attr}_k\}$ **do**
  if $\texttt{Attr}_i$ is numeric **then**
    $P \leftarrow$ create a partition space for $Attr_i$ with R partitions
    $P_L \leftarrow$ label P based on tuples T and regions A, N
    $P_{L,Filtered} \leftarrow$ filter P_L
    $P^* \leftarrow$ fill the gap in $P_{L,Filtered}$ based on δ
    Norm($\texttt{Attr}_i$) $\leftarrow$ Normalization of $\texttt{Attr}_i$ into $[0, 1]$ values
    $\mu_A \leftarrow$ Average of Norm($\texttt{Attr}_i$) for tuples in A
    $\mu_N \leftarrow$ Average of Norm($\texttt{Attr}_i$) for tuples in N
    $d \leftarrow |\mu_A - \mu_N|$
    if P^* contains a single block of consecutive abnormal partitions ***and*** $d > \theta$ **then**
      Pred $\leftarrow$ extract a candidate predicate from P^*
      $\pi \leftarrow \pi \cup$ Pred **// add Pred into the list**
    end
  else if $\texttt{Attr}_i$ is categorical **then**
    $P \leftarrow$ create a partition space for $Attr_i$ with $|Unique(\texttt{Attr}_i)|$ partitions
    $P_L \leftarrow$ label P based on tuples T and regions A, N
    if P_L has at least one abnormal partition **then**
      Pred $\leftarrow$ extract a candidate predicate from P_L
      $\pi \leftarrow \pi \cup$ Pred **// add Pred into the list**
    end
  end
end
return π

Algorithm 1: Predicate Generation.

and normal regions into the partition space, as shown in Figure 2.3. A secondary goal of our discretization step is to reduce the influence of having more tuples with normal values than with abnormal values. Thus, our discretization enables us to focus on the distribution of an attribute's value across the two regions.

Here, the number of partitions, R, is an important parameter, which decides the trade-off between our algorithm's computation time (see Section 2.4.6) and the ability of the individual par-

titions to distinguish normal from abnormal tuples. By default, DBSherlock uses 1,000 partitions (R=1,000) for numeric attributes, which is large enough to separate abnormal and normal tuples, yet small enough to optimize computation time.

For each categorical attribute $\texttt{Attr}_i$, we create $|Unique(\texttt{Attr}_i)|$ number of partitions. Here, $|Unique(\texttt{Attr}_i)|$ is the number of unique values found in our dataset for $\texttt{Attr}_i$, i.e., one partition per each value of the attribute. We use $\texttt{C}_j$ to denote the (categorical) value represented by partition $\texttt{P}_j$. Unlike numeric attributes, the order of partitions for categorical attributes is unimportant.

2.4.2 Partition Labeling

Once the partition space is created, the next step is to mark each partition with one of three labels: {Empty, Normal, Abnormal}. For a numeric attribute $\texttt{Attr}_i$, an input tuple belongs to partition $\texttt{P}_j$, if the tuple's value for $\texttt{Attr}_i$ lies within $\texttt{P}_j$'s boundaries. If every tuple belonging to $\texttt{P}_j$ lies in the abnormal region specified by the user, $\texttt{P}_j$ is labeled as Abnormal. Conversely, if every tuple belonging to $\texttt{P}_j$ lies in the normal region, $\texttt{P}_j$ is labeled as Normal. Otherwise, the partition label is left Empty. (See Figure 2.3.)

For a categorical attribute, an input tuple belongs to a partition $\texttt{P}_j$, if the tuple's value for $\texttt{Attr}_i$ equals the category value of the partition, i.e., $\texttt{Attr}_i = \texttt{C}_j$. Since our categorical attributes are less noisy, we use a simpler labeling strategy. Let $\texttt{P}_j\texttt{(A)}$ and $\texttt{P}_j\texttt{(N)}$ be the number of tuples belonging to $\texttt{P}_j$ in the abnormal and normal regions, respectively. $\texttt{P}_j$ is labeled as Abnormal if $\texttt{P}_j\texttt{(A)} > \texttt{P}_j\texttt{(N)}$. Similarly, $\texttt{P}_j$ is labeled as Normal if $\texttt{P}_j\texttt{(A)} < \texttt{P}_j\texttt{(N)}$. Otherwise, the partition label is left with an Empty label. For categorical attributes, our algorithm extracts a candidate predicate from the partition space right after the labeling step (see Section 2.4.5), skipping the next two steps.

2.4.3 Partition Filtering

After labeling, we filter some of the Normal and Abnormal partitions by replacing their labels with an Empty one. The filtering step is only applied to our numeric attributes, which are quite noisy. During this step, a partition $\texttt{P}_j$'s label is replaced with Empty, if its original label is different from either of its two non-Empty adjacent partitions (i.e., the closest non-Empty partitions on the left and right side of $\texttt{P}_j$ in the partition space). Figure 2.4 demonstrates various cases where a partition $\texttt{P}_j$ is filtered out. Note that the only case where $\texttt{P}_j$ remains unchanged is when both of its non-Empty adjacent partitions have the same label as $\texttt{P}_j$ itself (shown as Scenario 1 in Figure 2.4). If we only have a single Normal or Abnormal partition to begin with, we deem it significant and do not filter it. Once all non-Empty partitions are processed, their labels are changed to Empty *simultaneously*. We do not perform this procedure incrementally to prevent a situation where partitions continuously

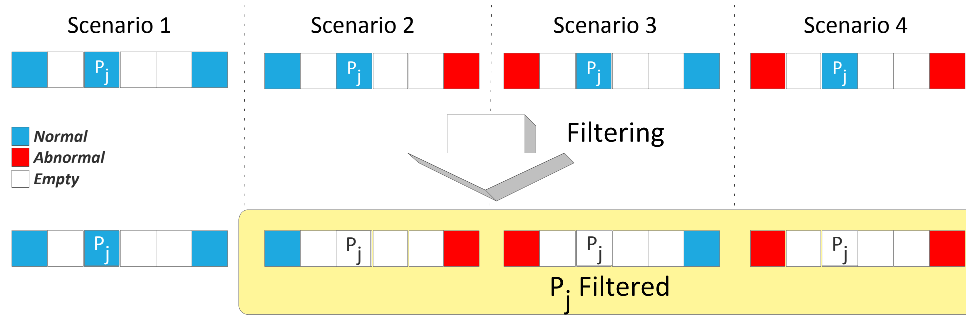


Figure 2.4: Different scenarios of filtering P_j in the partition space.

filter each other out. If filtering is performed incrementally, the two partitions at each end of the partition space will also get filtered in Scenarios 2 and 3 in Figure 2.4.

Our filtering strategy aims to separate the `Normal` and `Abnormal` partitions that are originally mixed across the partition space (e.g., due to the noise in the data or user errors). If there is a predicate on attribute $\mathtt{Attr}_i$ that has high separation power, the `Normal` and `Abnormal` partitions are very likely to form well-separated clusters after the filtering step. This step mitigates some of the negative effects of noisy data or user's error, which could otherwise exclude a predicate with high separation power from the output. This idea is visually shown in Figure 2.3.

2.4.4 Filling the Gaps

This step is only applied to numeric attributes. After the filtering step, there will be larger blocks of consecutive `Abnormal` and `Normal` partitions, separated by `Empty` partitions. These `Empty` partitions were either initially `Empty` or were filtered out during the filtering step. Our algorithm fills these gaps before the predicate generation by labeling these `Empty` partitions as either `Normal` or `Abnormal` as follows.

We compare the distance of each `Empty` partition P_j to its two adjacent non-`Empty` partitions. If both adjacent partitions have the same label or if P_j has only one adjacent non-`Empty` partition (i.e., j=1 or j=R), P_j will receive the same label as its adjacent non-`Empty` partition(s) at the end of this step. If P_j's two adjacent non-`Empty` partitions have different labels, we calculate P_j's distance to each partition and assign it the label of the closer partition.

There is a special case where only `Abnormal` partitions remain after the filtering step. In this case, if we naïvely fill the gaps, every partition will become `Abnormal` and the algorithm will not find any predicates for the attribute. To handle this special case, we calculate the average value of the attribute over the input tuples in the normal region and label the partition that contains this average value as `Normal` regardless of its previous label. Then we fill the gaps according to the

previously described procedure. (Without any `Normal` partitions, we will not be able to determine the direction of the predicate in the next step of the algorithm, i.e., whether $\texttt{Attr}_i < \texttt{v}$ or $\texttt{Attr}_i > \texttt{v}$.)

To control the behavior of our algorithm, we also introduce a parameter δ, called the *anomaly distance multiplier*. When the `Empty` partition $\texttt{P}_j$ is processed by the above procedure, we multiply its distance to its adjacent `Abnormal` partition by δ. Thus, $\delta{>}1$ will cause more `Empty` partitions to be labeled as `Normal` while $\delta{<}1$ results in more `Empty` partitions being labeled as `Abnormal`. In other words, with parameter δ, we can tune our predicates: $\delta{<}1$ for more general predicates (i.e., more likely to flag tuples as abnormal) and $\delta{>}1$ for more specific ones (i.e., less likely to flag tuples as abnormal). By default, `DBSherlock` uses $\delta = 10$.

2.4.5 Extracting Predicates from Partitions

This step is applied to both numeric and categorical attributes using slightly different procedures. For numeric attributes, the previous filtering step allows us to find attributes that have a predicate with high separation power, but there is still a possibility that some of these attributes are not related to the actual cause of the anomaly. To mitigate this problem, we perform the following procedure. First, we normalize each numeric attribute $\texttt{Attr}_i$ by subtracting its minimum value from its original values $\texttt{val}_i$ and dividing them by the attribute's range:

$$\texttt{Norm}(\texttt{val}_i) = \frac{\texttt{val}_i - \texttt{Min}(\texttt{Attr}_i)}{\texttt{Max}(\texttt{Attr}_i) - \texttt{Min}(\texttt{Attr}_i)} \tag{2.2}$$

This results in the values of an attribute to range in $[0, 1]$. Let μ_A and μ_N be the average values of Norm($\texttt{Attr}_i$) for tuples in the abnormal and normal regions, respectively. `DBSherlock` extracts a candidate predicate from $\texttt{Attr}_i$'s partition space only if $|\mu_A - \mu_N| {>} \theta$, where θ is a parameter called the *normalized difference threshold*. The user can tune this threshold to adjust the selectivity of `DBSherlock` in finding predicates

After performing these normalization and thresholding procedures, we can extract candidate predicates from the partition space, as follows. As noted in section 2.3, we only seek predicates of the form $\texttt{Attr}_i < \texttt{x}$, $\texttt{Attr}_i > \texttt{x}$, and $\texttt{x} < \texttt{Attr}_i < \texttt{y}$. In the partition space, these types of predicates correspond to a single block of consecutive `Abnormal` partitions. Therefore, we extract a candidate predicate `Pred` for an attribute $\texttt{Attr}_i$ if and only if there is a single block of consecutive `Abnormal` partitions.

For categorical attributes, our procedure for extracting a candidate predicate is much simpler. `DBSherlock` traverses the partition space of such attributes and extracts each category value $\texttt{C}_j$ if its partition $\texttt{P}_j$ is labeled as `Abnormal`. A predicate for a categorical attribute is of the form

`Attr`$_i \in \{c_1, \ldots, c_l\}$, where l is the number of partitions labeled as `Abnormal` and c_i's are their corresponding category values.

2.4.6 Time Complexity

For each attribute, our predicate generation algorithm scans the input tuples to label each partition. Then, it iterates over these partitions twice in the subsequent steps (i.e., 'Filtering' and 'Filling the gap' in Figure 2.3). Thus, the time complexity of `DBSherlock`'s predicate generation algorithm is $O(k(X+R))$, where R is the number of partitions,[6] X is the number of input tuples, and k is the number of attributes.

2.5 Incorporating Domain Knowledge

Our algorithm extracts predicates that have a high diagnostic power (see Section 2.8). However, some of these predicates may be secondary symptoms of the root cause, which if removed, can make the diagnosis even easier. This is because the fact that `Pred`$_i$ **implies** an anomaly, we cannot conclude that it also **causes** it. In fact, there could be another predicate, say `Pred`$_j$, causing both `Pred`$_i$ and the anomaly. While many existing algorithms [86, 172, 200, 201, 220] learn causal relationships by verifying whether the significant association between two variables persistently holds, `DBSherlock` cannot simply use such algorithms to generate predicates or prune secondary symptoms, because they are applicable only when there are large enough data of database performance anomalies available to learn causal relationships. Unfortunately, large enough data of database performance anomalies are not readily available in academia. Therefore, `DBSherlock`'s algorithm is developed so that it does not have to rely on having large training data of database performance anomalies.

Thus, to further improve the accuracy of our predicates and prune secondary symptoms, `DBSherlock` allows for incorporating domain knowledge of attributes' semantics into the system. However, note that this mechanism is an optional feature, and as we show in our experiments, `DBSherlock` produces highly accurate explanations even without any domain knowledge (see Section 2.8.3). Also, `DBSherlock` is bootstrapped with domain knowledge only once for each specific version of OS or DBMS. In other words, DBAs do not need to modify this, as the semantics of DBMS and OS variables do not depend on the workload, e.g., `OS CPU Usage` always has the same meaning regardless of the specific workload.

Every piece of domain knowledge is encoded as a rule: `Attr`$_i \rightarrow$ `Attr`$_j$. Each rule must satisfy the following conditions:

[6] A similar analysis applies if the number of partitions differ.

i. If predicates $\texttt{Pred}_i$ and $\texttt{Pred}_j$ (corresponding to attributes $\texttt{Attr}_i$ and $\texttt{Attr}_j$, respectively) are both extracted, $\texttt{Pred}_j$ is likely to be a secondary symptom of $\texttt{Pred}_i$.

ii. $\texttt{Attr}_i \rightarrow \texttt{Attr}_j$ and $\texttt{Attr}_j \rightarrow \texttt{Attr}_i$ cannot coexist.

For instance, if $\texttt{Attr}_i$ is the 'DBMS CPU Usage' and $\texttt{Attr}_j$ is the 'OS CPU Usage', then $\texttt{Attr}_i \rightarrow \texttt{Attr}_j$ is a valid rule since DBMS CPU usage effects OS CPU usage, but not vice versa.

However, given a rule $\texttt{Attr}_i \rightarrow \texttt{Attr}_j$, and the corresponding predicates $\texttt{Pred}_i$ and $\texttt{Pred}_j$, whether $\texttt{Pred}_j$ will be filtered out will require further analysis, since the domain knowledge may not be a perfect reflection of the reality either. For instance, the rule 'DBMS CPU Usage' $\rightarrow$ 'OS CPU Usage' may occasionally break. For example, there might be other attributes, such as 'Number of Processes' or 'Number of Threads' that are not utilized by the DBMS, but may affect 'OS CPU Usage'.

As a solution, DBSherlock tests the independence between $\texttt{Attr}_i$ and $\texttt{Attr}_j$ based on their continuous (or categorical) values. For continuous attributes, we discretize the two attributes $\texttt{Attr}_i$ and $\texttt{Attr}_j$ with γ equi-width bins for each attribute. We then construct a two-dimensional joint histogram from the input data, estimating the joint probability distribution of the two attributes. For categorical attributes, a joint histogram is constructed from the input data. For testing independence, we use the joint probability distribution of the two attributes to calculate their *mutual information*.

We denote the mutual information of two attributes $\texttt{Attr}_i$ and $\texttt{Attr}_j$ by $\texttt{MI}(\texttt{Attr}_i, \texttt{Attr}_j)$, defined as:

$$\texttt{MI}(\texttt{Attr}_i, \texttt{Attr}_j) = H(\texttt{Attr}_i) + H(\texttt{Attr}_j) - H(\texttt{Attr}_i, \texttt{Attr}_j)$$

where $H(\texttt{Attr}_i)$ is the entropy of the attribute $\texttt{Attr}_i$ and $H(\texttt{Attr}_i,\ \texttt{Attr}_j)$ is the joint entropy of the two attributes [88]. An independence factor $\kappa(\texttt{Attr}_i, \texttt{Attr}_j)$ of the two attributes is then calculated as follows:

$$\kappa(\texttt{Attr}_i, \texttt{Attr}_j) = \frac{\texttt{MI}(\texttt{Attr}_i, \texttt{Attr}_j)^2}{H(\texttt{Attr}_i) H(\texttt{Attr}_j)}$$

The value of κ will be 0, if the two attributes are independent and approaches 1 with higher dependence. We perform the independence test by comparing the value of κ with a threshold κ_t (by default, we use $\kappa_t = 0.15$), and the two attributes pass the test if $\kappa < \kappa_t$. If the two attributes do not pass the independence test, we conclude that the rule $\texttt{Attr}_i \rightarrow \texttt{Attr}_j$ is indeed valid, and $\texttt{Pred}_j$ is merely a secondary symptom of $\texttt{Pred}_i$ and filter out $\texttt{Pred}_j$. If the two attributes pass the independence test, we conclude that the rule $\texttt{Attr}_i \rightarrow \texttt{Attr}_j$ does not apply, and leave both predicates in the output.

For MySQL on Linux, the following four rules are sufficient to encode such relationships:

1. DBMS CPU Usage $\rightarrow$ OS CPU Usage

2. `OS Allocated Pages` $\rightarrow$ `OS Free Pages`

3. `OS Used Swap Space` $\rightarrow$ `OS Free Swap Space`

4. `OS CPU Usage` $\rightarrow$ `OS CPU Idle`

The first rule encodes the subset relationship. The last three rules encode the fact that one attribute is always a constant value minus the other attribute, and is thus uninteresting. In Section 2.8.6, we show that even without these rules, `DBSherlock`'s accuracy drops by only 2–3%.

2.6 Incorporating Causal Models

Previous work on performance explanation [155] has only focused on generating explanations in the form of predicates. `DBSherlock` improves on this functionality by generating substantially more accurate predicates (20-55% higher F1-measure; see Section 2.8.4). However, a primary objective of `DBSherlock` is to go beyond raw predicates, and offer explanations that are more human-readable and descriptive. For example, the cause of a performance hiccup could be a network congestion due to a malfunctioning network router. Initially, the user will rely on `DBSherlock`'s generated predicates as diagnostic clues to identify the actual cause of the performance problem more easily. However, once the root cause is diagnosed, s/he can notify `DBSherlock` as to what the actual cause was. `DBSherlock` then relates the generated predicates to the actual cause and saves them in the form of a causal model. This model will be consulted in future diagnoses to provide a human-readable explanation (i.e., 'malfunctioning router') for similar situations.

To utilize the user feedback, `DBSherlock` uses a simplified version of the *causal model* proposed in the seminal work of Halpern and Pearl [129]. Our causal model consists of two parts: *cause variable* and *effect predicates*. The *cause variable* is a binary, exogenous variable[7] labeled by the end user. When the *cause variable* is set to *true*, it activates all of its *effect predicates*. For example, Figure 2.5 is a causal model with 'Log Rotation' as the *cause variable* and three *effect predicates*. According to this model, if there is an event of 'Log Rotation' (i.e. *cause variable* is *true*) then these three *effect predicates* will also be true.

The following example describes how such causal models are constructed and used in `DBSherlock`. Consider a scenario where the user selects an abnormal region for which `DBSherlock` returns the following predicates:

$$\texttt{CpuWait} > 50\% \ \land \ \texttt{Latency} > 100\text{ms} \ \land \ \texttt{DiskWrite} > 5\text{MB/s}$$

[7] An exogenous variable is a variable whose values are determined by factors outside the model [129].

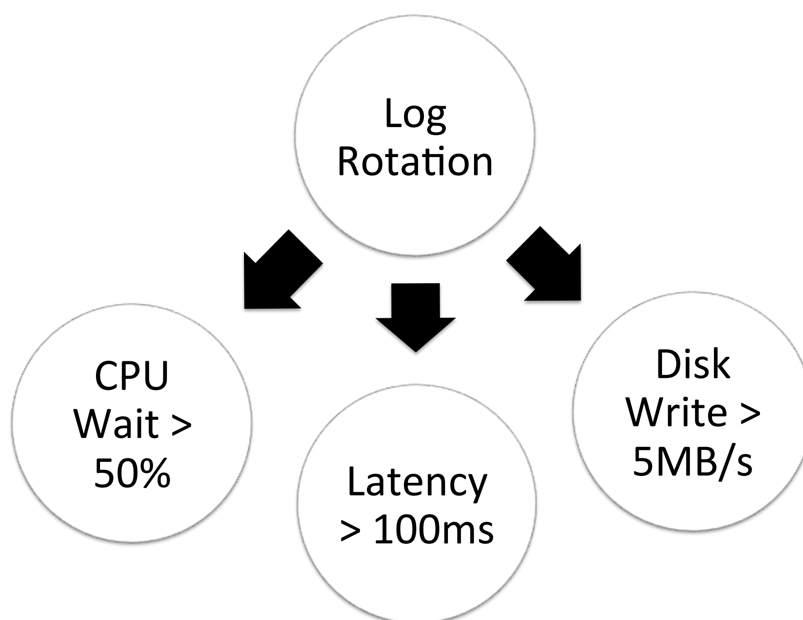


Figure 2.5: An example of a causal model in `DBSherlock`.

Also, suppose that the user is able to diagnose the actual cause of the problem with the help of these predicates; assume that by inspecting the recent system logs she establishes that the cause was the rotation of the redo log file.[8] Once this feedback is received from the user, `DBSherlock` can link this cause with these predicates as shown in Figure 2.5. After that, for every diagnosis inquiry in the future, `DBSherlock` calculates the *confidence* of this causal model and 'Log Rotation' is reported as a possible cause if its confidence is higher than a threshold λ. By default, `DBSherlock` displays only those causes whose confidence is higher than λ=20%. However, the user can modify λ as a knob (i.e., using a sliding bar) to interactively view fewer or more causes.

Over time, additional causal models might be added in the system as a result of inspecting new performance problems. When multiple causal models are available in the system, `DBSherlock` consults all of them (i.e., computes their confidence) and returns those models whose confidence is higher than λ to the user, presented in their decreasing order of confidence. When none of the causes offered by the existing models are deemed helpful by the user, she always has the option of asking `DBSherlock` to simply show the original predicates instead. Also, when none of the causal models achieve a confidence higher than λ (e.g., when the given anomaly has not been previously observed in the system), `DBSherlock` only displays the generated predicates. Again, once the cause is diagnosed and shared with the system, `DBSherlock` creates a new causal model to be used in the

[8]In MySQL, log rotations can cause performance hiccups when the adaptive flushing option is disabled.

future. (See Figure 2.1 in Section 2.2.)

We evaluate the accuracy of the generated explanations in Section 2.8. Next, we explain how DBSherlock computes the confidence of each causal model (Section 2.6.1), and how it merges multiple models to improve their explanatory power (Section 2.6.2).

2.6.1 Confidence of a Causal Model

In DBSherlock we define the confidence of a causal model as the average separation power of its *effect predicates* in the partition space. Note that, unlike equation (2.1), here we use the partition space instead of the input tuples to reduce the effect of the noise in real-world data (see Section 2.4.1). Formally:

Confidence of a causal model. Let $\{\mathtt{Pred}_1, \ldots, \mathtt{Pred}_n\}$ be the *effect predicates* of a given causal model $\mathcal{M}$, where $\mathtt{Pred}_i$ is defined over $\mathtt{Attr}_i$. Also, let $P_{i,N}$ and $P_{i,A}$ be the partitions labeled as Normal and Abnormal in the partition space of $\mathtt{Attr}_i$, respectively. We define the *confidence* $\mathcal{C}_{\mathcal{M}}$ of the causal model $\mathcal{M}$ as:

$$\mathcal{C}_{\mathcal{M}} = \frac{\sum_{i=1}^{n} \frac{|\mathtt{Pred}_i(P_{i,A})|}{|P_{i,A}|} - \frac{|\mathtt{Pred}_i(P_{i,N})|}{|P_{i,N}|}}{n} \tag{2.3}$$

where $\mathtt{Pred}(P)$ is the set of partitions in P that satisfy predicate Pred.

The idea behind this definition is to estimate the likelihood of a *cause variable* being true given the normal and abnormal partitions, based on the assumption that if the model's *cause variable* is true, then its *effect predicates* are also likely to exhibit high separation power in their partition spaces.

2.6.2 Merging Causal Models

Among the *effect predicates* of a causal model, some predicates may have less or no relevance to the actual cause, e.g., some predicates could simply be a side-effect of the actual cause. Also, since the *effect predicates* of a single causal model reflect the specific values observed in a particular instance of an anomaly, they may not be applicable to other instances of the same cause. In DBSherlock, multiple causal models might be created for the same cause while analyzing different anomalies over time. DBSherlock can improve such causal models by merging them into a single one.

Merging causal models eliminates some of the unnecessary and less relevant *effect predicates*, while enabling relevant *effect predicates* to apply to different anomaly instances caused by the same cause. We merge two causal models by:

1. keeping only those *effect predicates* that are on attributes common to both models; and

2. merging two predicates on the same attribute into a single predicate that includes the boundaries (or categories) of both.

Suppose that we have two causal models with the same cause: $\mathcal{M}_1$ with the *effect predicates* {A > 10, B > 100, C > 20, E ∈ {'xx', 'yy', 'zz'}}, and $\mathcal{M}_2$ with the *effect predicates* {A > 15, C > 15, D < 250, E ∈ {'xx', 'zz'}}. To merge $\mathcal{M}_1$ and $\mathcal{M}_2$, we only keep {A > 10, C > 20, E ∈ {'xx', 'yy', 'zz'}} from $\mathcal{M}_1$ and {A > 15, C > 15, E ∈ {'xx', 'zz'}} from $\mathcal{M}_2$ since attributes A, C and E are common to both models.

Next, we compare the two predicates on the same attribute and merge them such that the merged predicate includes both. Here, merging {A > 10} and {A > 15} leads to {A > 10} and merging {C > 20} and {C > 15} leads to {C > 15}. Likewise, merging {E ∈ {'xx', 'yy', 'zz'}} and {E ∈ {'xx', 'zz'}} leads to {E ∈ {'xx', 'zz'}}. Thus, in this example, the *effect predicates* of the merged causal model will be {A > 10, C > 15, E ∈ {'xx', 'yy', 'zz'}}. Note that numeric predicates with different directions (e.g., {A > 10} and {A < 30}) are considered inconsistent. Such predicates are not merged and will be discarded.

We study the effect of merging causal models in Section 2.8.5, where we show that the merged causal models are on average 30% more accurate than the original models.

2.7 Automatic Anomaly Detection

Sometimes, an anomaly may not be visually obvious to a human user inspecting a performance plot. In such situations, users may mistakenly specify a normal region as abnormal and vice versa. To aid with these cases, DBSherlock also provides an option for automatic anomaly detection. Thus, users can either (i) rely on DBSherlock to find and suggest anomalies to them, or (ii) continue to manually find anomalies but compare them with those found by DBSherlock for reassurance.

There is much work on outlier detection in different contexts [64, 96, 154, 212, 213, 214, 226, 238, 239, 242, 249]. In DBSherlock, we introduce an algorithm for the automatic detection of the anomaly regions. Our algorithm utilizes the DBSCAN clustering algorithm [108] and works as follows.

First, we normalize each attribute $\texttt{Attr}_i$, which is equivalent to the normalization step in our predicate generation algorithm (Equation (2.2) in Section 2.4.5). We then choose relevant attributes to detect possible anomalies, which are characterized by a subsequence in the time series with an abrupt change in the values. For attributes that we cannot identify such a behavior, we exclude them from our analysis as they are likely to have an insignificant separation power. We quantify this behavior and call it a *potential power* of an attribute, denoted as $PP(\texttt{Attr}_i)$.

To calculate $PP(\texttt{Attr}_i)$, we first define a sliding window $w(\tau)$ as a subsequence of size τ in the time series. We also denote the median of $\texttt{Attr}_i$ as $\texttt{Median}(\texttt{Attr}_i)$ and denote the median of

the values within a sliding window $w(\tau)$ as $\texttt{Median(Attr}_i\texttt{, } w\texttt{)}$. Then $PP(\texttt{Attr}_i)$ is calculated as follows:

$$PP(\texttt{Attr}_i) = \max_{w \in W} |\texttt{Median(Attr}_i\texttt{)} - \texttt{Median(Attr}_i\texttt{, } w\texttt{)}| \tag{2.4}$$

where W represents the set of all possible sliding windows of size τ. Equation (2.4) uses a median filter to calculate the maximum absolute difference between the overall median and the median of values in each window. We only include attributes with a potential power greater than a threshold $PP_t \in [0, 1]$. (DBSherlock uses $\tau = 20$ and $PP_t = 0.3$ as default values.)

We use DBSCAN to build clusters with the selected attributes from the previous step. DBSCAN takes two parameters, ϵ and $minPts$. For our algorithm, we fix $minPts$ to 3 and use the *k-dist* function to build a list L_k of the distances of the *k-th nearest neighbors*, as suggested in [108], to determine ϵ. We have empirically found $\epsilon = \max(L_k)/4$ to perform well in DBSherlock.

Given the clusters formed by DBSCAN, our algorithm returns the points in all clusters whose sizes are less than 20% of the total number of data items. This is under the assumption that the abnormal region is relatively smaller than the normal region.

2.8 Evaluation

In this section, we empirically evaluate the effectiveness of DBSherlock. The goals of our experiments are to show that:

(i) Our causal models produce accurate explanations (Section 2.8.3).

(ii) Even without causal models, the raw predicates generated by DBSherlock are more accurate than those generated by the state-of-the-art explanation framework (Section 2.8.4).

(iii) Our idea of merging causal models improves the quality of our explanations significantly (Sections 2.8.5).

(iv) Incorporating domain knowledge allows DBSherlock to achieve higher accuracy (Section 2.8.6).

(v) DBSherlock is able to explain compound situations where multiple anomalies arise simultaneously (Section 2.8.7).

(vi) Using our predicates, users can diagnose the actual cause of performance anomalies much more accurately (Section 2.8.13).

2.8.1 Experiment Setup

To collect log data with different types of anomalies, we ran different mixtures of the TPC-C benchmark [23] on Microsoft Azure [6] virtual machine instances. In all our experiments, we have

used two Microsoft Azure A3-tier instances, each with 4 CPU cores of 2.1Ghz (AMD Opteron 4171H) and 7GB of RAM running Ubuntu 14.04. We employed one of the two A3 instances to run MySQL 5.6.20 and the other to simulate clients (using OLTPBenchmark framework [7, 99]). For stress-based experiments, we also used a tool called stress-ng [11] which can artificially stress the system by taking up excessive CPU, I/O and network resources when needed. Each individual experiment (called a *dataset*) consisted of two minutes of normal activity plus one or more abnormal situations (of varying length). We ran our experiments using TPC-C. The default setting used in our TPC-C workload was a scale factor of 500 (i.e., 50GB) with 128 terminals. We also experimented with different scale factors (from 16 to 500) and number of terminals (from 16 to 128). The results were consistent across these different settings, and thus we only report our results using the default setting described above. In each dataset, we intentionally created various abnormal situations on the server, as described next.

Type of anomaly	**Description**
Poorly Written Query	Execute a poorly written JOIN query, which would run efficiently if written properly.
Poor Physical Design	Create an unnecessary index on tables where mostly INSERT statements are executed.
Workload Spike	Greatly increase the rate of transactions and the number of clients simulated by OLTPBenchmark (128 additional terminals with transaction rate of 50,000).
I/O Saturation	Invoke stress-ng, which spawns multiple processes that spin on write()/unlink()/sync() system calls.
Database Backup	Run *mysqldump* on the TPC-C database instance to dump the table to the client machine over the network.
Table Restore	Dump the pre-dumped *history* table back into the database instance.
CPU Saturation	Invoke stress-ng, which spawns multiple processes calling poll() system calls to stress CPU resources.
Flush Log/Table	Flush all tables and logs by invoking *mysqladmin* commands ('flush-logs' and 'refresh').
Network Congestion	Simulate network congestion by adding an artificial 300-milliseconds delay to every traffic over the network via Linux's *tc* (Traffic Control) command.
Lock Contention	Change the transaction mix to execute *NewOrder* transactions only on a single warehouse and district.

Table 2.1: Ten types of performance anomalies used in our experiments.

2.8.2 Test Cases

To test our algorithm, we created 10 different classes of anomalies to represent some of the important types of real-world problems that can deteriorate the performance of a database. During the two-minute run of the normal workload in each dataset, we invoked the actual cause of an anomaly with different start times and durations. For each type of anomaly, we collected 11 different datasets by varying the duration when possible (e.g., stressing system resources) or its start time (i.e., the time when the cause of an anomaly is triggered) when the actual duration was impossible to control (e.g., running *mysqldump*). The duration or start time of the anomalies ranged from 30 to 80 seconds with the increment of 5 seconds, yielding 11 datasets (i.e., 30, 35, $\cdots$, 80) for each type of anomaly (a total of 110 datasets). For each dataset, we manually selected a region of anomaly via visual inspection; the region left unselected automatically became the normal region.

Table 2.1 lists the types and descriptions of the different classes of anomalies that we tested within our experiments. These anomalies are designed to reflect a wide range of realistic scenarios that can negatively impact the performance of a transactional database.

2.8.3 Accuracy of Single Causal Models

Our goal in this section is to evaluate the effectiveness of our causal models in producing correct explanations. It is quite common that an anomaly from a certain cause is only observable a few times over the lifetime of a database operation. This makes log samples of such anomalies quite scarce in many cases (e.g., disk failure) and thus necessitates that our framework identifies the correct cause even when our causal model is created from a single dataset. Thus, in each test case we only used a single dataset to construct a causal model with θ=0.2 (which is the normalized difference threshold, see Section 2.4.5). This is the default value of θ in `DBSherlock` chosen to aggressively filter out attributes with insignificant behavior in the anomaly region. We applied the constructed causal model on all the remaining 109 datasets to obtain its confidence in each test case. We repeated this process until every dataset was chosen to construct a causal model.

With our algorithm, the correct causal models achieve the highest confidence in all 10 test cases (i.e., the correct cause was shown as the most likely cause to the user). The margin (i.e., positive difference) of confidence between the correct model and the highest among incorrect models is on average 13.5%. In other words, not only does the correct model achieve the highest confidence (and is shown to the user as the most likely cause), but its confidence is also well separated from the highest-ranked incorrect model. (In Section 2.8.5, we show that our model merging technique improves this margin even further.) Figure 2.6 shows the margin of confidence of the correct causal model, which compares the average confidence of the correct causal model to the highest confidence among all other (incorrect) models for different types of anomalies.

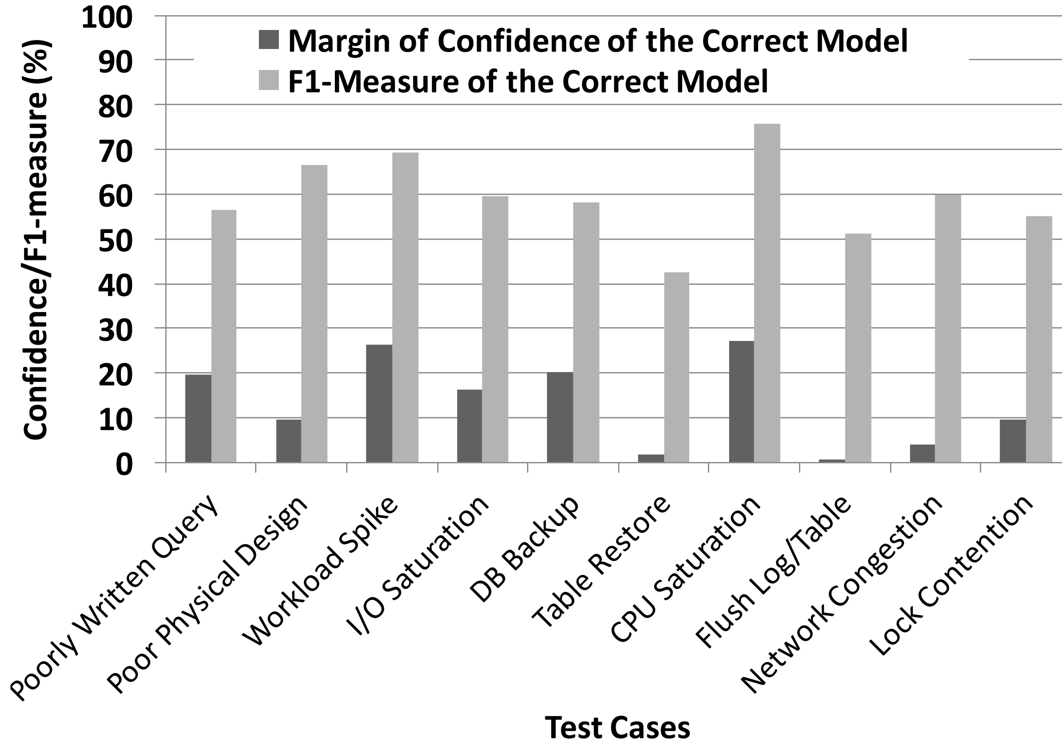


Figure 2.6: The margin of confidence and the average F1-measure of the correct causal model for different anomalies.

Here, anomalies caused by 'Table Restore' and 'Flush Log/Table' proved most illusive, as they yielded the lowest confidence among the causal models. This was because the two anomalies shared the common characteristic that the DBMS performed too many disk I/Os. However, even in this case, `DBSherlock` could correctly distinguish the correct cause from the incorrect ones.

Overall, this challenging experiment is an extremely encouraging result showing that `DBSherlock` is capable of generating the correct explanation even with a single dataset as a training sample and in the presence of 9 other competing models.

2.8.4 DBSherlock Predicates versus PerfXplain

We compared the accuracy of our predicates with predicates generated by the state-of-the-art performance explanation framework, PerfXplain [155]. Since PerfXplain is designed to work with MapReduce logs, we had to re-implement PerfXplain's algorithm to fit into our context. Originally, PerfXplain operates on pairs of MapReduce jobs. Instead, we modified it to use pairs of our input tuples. We used the following query for PerfXplain:

`EXPECTED avg_latency_difference = insignificant`

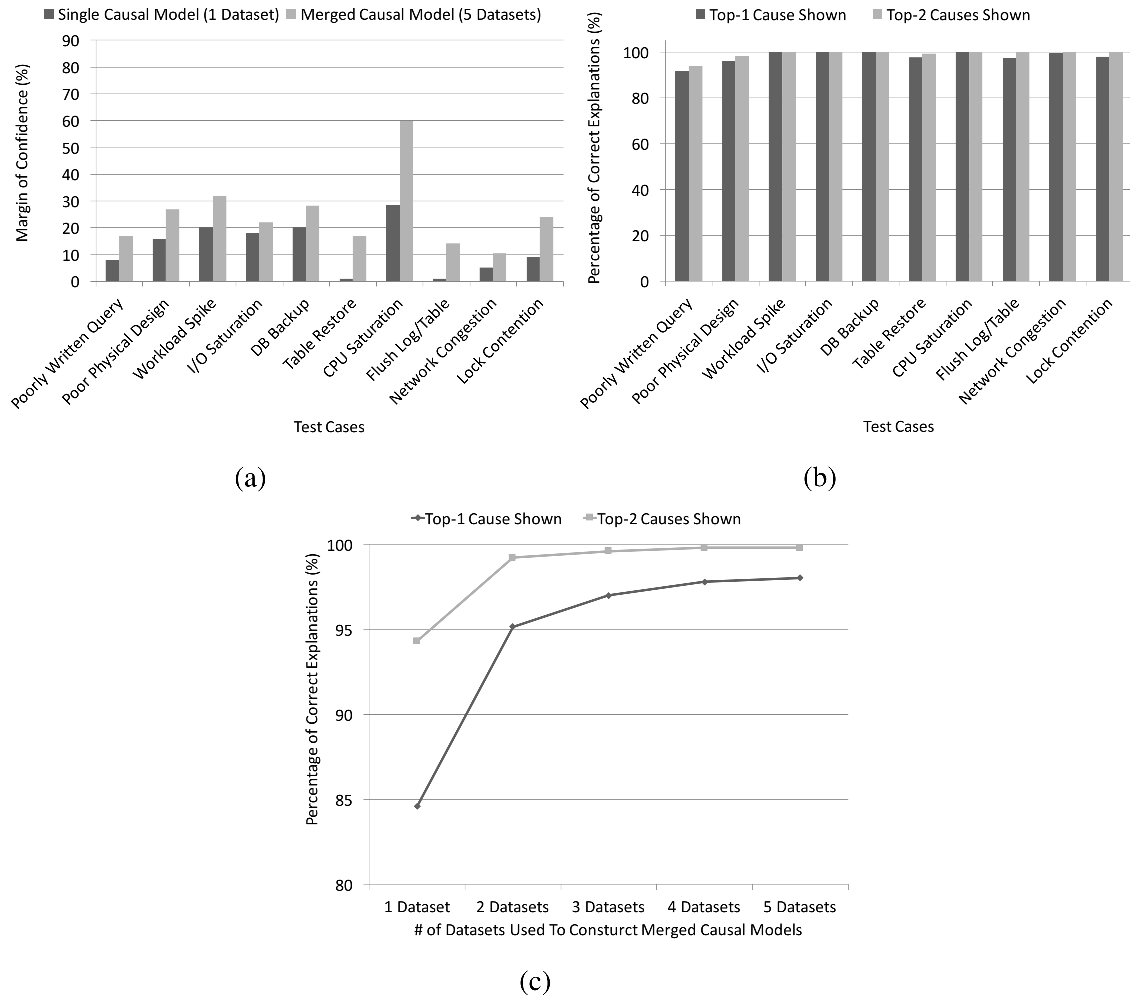


Figure 2.7: (a) The margin of confidence for single versus merged causal models, (b) the ratio of correct explanations for merged causal models (if the top-k possible causes are shown to the user), and (c) the effect of the number of datasets (i.e., number of manual diagnoses required) on the accuracy of the casual model.

`OBSERVED avg_latency_difference = significant`

where two average latencies are deemed significant if their difference is at least 50% of the smaller value. We chose the same parameters for PerfXplain as suggested in [155] (i.e., we used 2,000 samples and a weight value of 0.8 for its scoring rules). We also varied the number of predicates from 1 to 10 and chose 2, which yielded the best results for PerfXplain. With 11 datasets for each case, we used 10 datasets to generate predicates and the accuracy of generated predicates was tested on the remaining dataset. Figure 2.8 demonstrates the average precision, recall and F1-measure in comparison. Our predicates achieved better accuracy than PerfXplain in nearly all cases (except for recall on one test case). Most notably, DBSherlock improves on PerfXplain's F1-

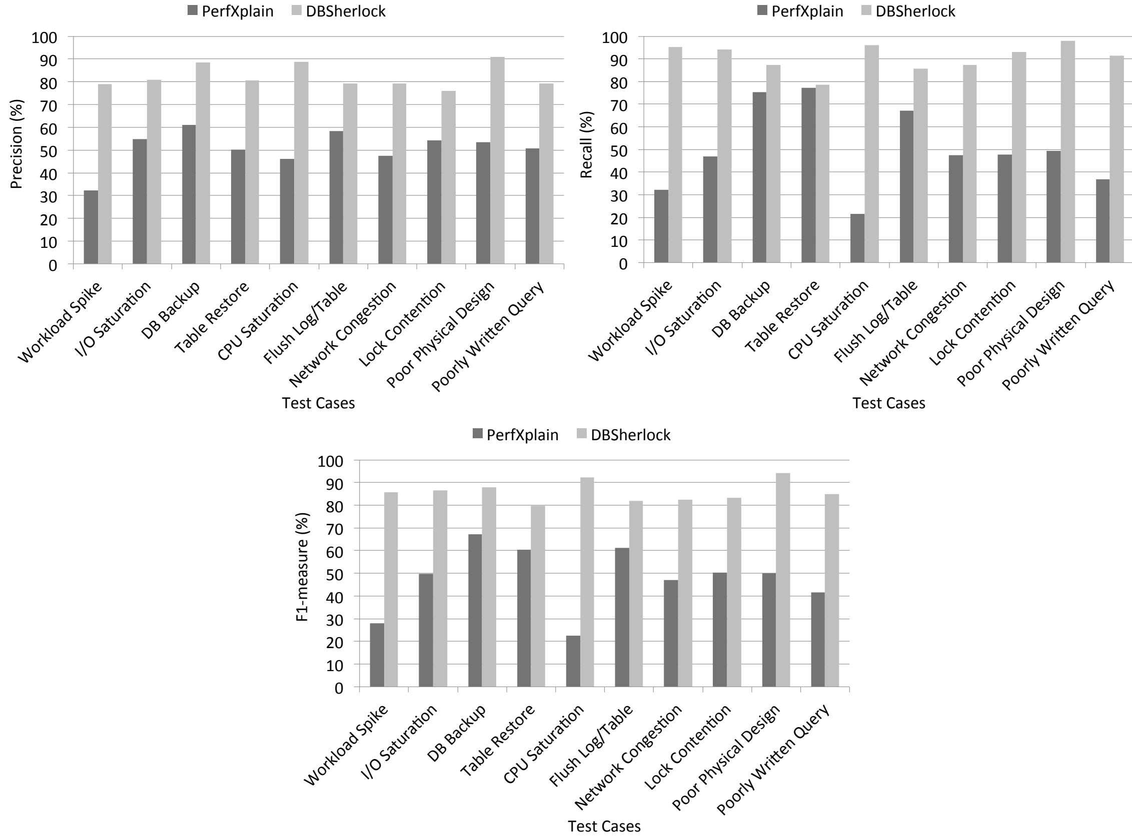


Figure 2.8: Average precision, recall and F1-measure of predicates generated by `DBSherlock` and PerfXplain.

measure by upto 55% (28% on average). This shows that performance explanation and diagnosis for OLTP workloads requires drastically different techniques than those developed for OLAP and map-reduce workloads.

2.8.5 Effectiveness of Merged Causal Models

When multiple causal models are available (from diagnosing different datasets), `DBSherlock` tries to merge them as much as possible in order to further improve the relevance and accuracy of the generated explanations. To evaluate the effectiveness of our merging strategy, we conducted a series of experiments, each using multiple datasets as training samples. We randomly assigned about 50% of the datasets from each type of anomaly (i.e., 5 out of 11 datasets) to construct and merge causal models for each type. Merged causal models were then used to calculate the confidence on the remaining 6 datasets. The process was repeated 50 times, resulting in 300 instances of explanations for each test case. We used a lower value of θ, namely $\theta = 0.05$, for our merged causal models (in contrast to 0.20 used for our single causal models). With a lower value

of θ, we can maximize the effect of merging causal models by having more predicates for each causal model at the start.

The results of these experiments are summarized in Figure 2.7a, showing that merging significantly increases the average margin of confidence against the second-highest confidence in all test cases.

To compare the accuracy of each explanation, we counted the number of cases where the correct cause was included in the top-k possible causes shown to a user. As shown in Figure 2.7b, `DBSherlock` presented the correct cause as the top explanation in almost every instance. In other words, k=1 would always be sufficient for achieving an accuracy greater than 98%. If we allow `DBSherlock` to list the top-2 possible explanations, then it identifies the correct cause in 99% of the cases.

We also studied the effectiveness of merging causal models with respect to the number of datasets used in constructing each causal model. As shown in Figure 2.7c, the accuracy of the merged causal models increases with more datasets. The accuracy quickly reaches 95% with only two datasets if only the top cause is returned, and it reaches 99% if the top two causes are returned. This experiment highlights that `DBSherlock` only requires a few manual diagnoses of an anomaly to construct highly accurate causal models.

In summary, the merging of causal models greatly improves the accuracy and quality of our explanations, generating predicates that are more relevant to the cause. Only a few datasets of the same anomaly are needed to construct a merged causal model that achieves an accuracy greater than 95%.

2.8.6 Effect of Incorporating Domain Knowledge

To study the effect of incorporating domain knowledge, we incorporated the four rules introduced in Section 2.5 into `DBSherlock`, and constructed single causal models with and without these rules, similar to the setup of Section 2.8.3.

	Accuracy if shown top-1 cause	**Accuracy if shown top-2 causes**
With Domain Knowledge	85.3%	94.8%
Without Domain Knowledge	82.7%	93.2%

Table 2.2: Ratio of correct causes with & without domain knowledge.

Table 2.2 reports the accuracy of single causal models with and without domain knowledge. Domain knowledge removed predicates that were a secondary symptom, and thus less relevant for the correct diagnosis of a given anomaly, improving the accuracy of causal models by 2.6%

if shown the top-1 cause and 1.6% if shown the top-2 causes. On the other hand, this experiment shows that, in practice, `DBSherlock` works surprisingly well even without any domain knowledge.

2.8.7 Explaining Compound Situations

It is not uncommon in a transactional database that multiple problems occur simultaneously. These compound situations add a new level of difficulty to diagnostic systems. We ran an experiment to address the framework's capability in such compound situations. We created six cases, where two or three anomalies happen at the same time during the two-minute run of our normal workload. For this experiment, causal models were constructed for each individual test case by merging causal models from every dataset. Explanations were then generated for the compound test cases.

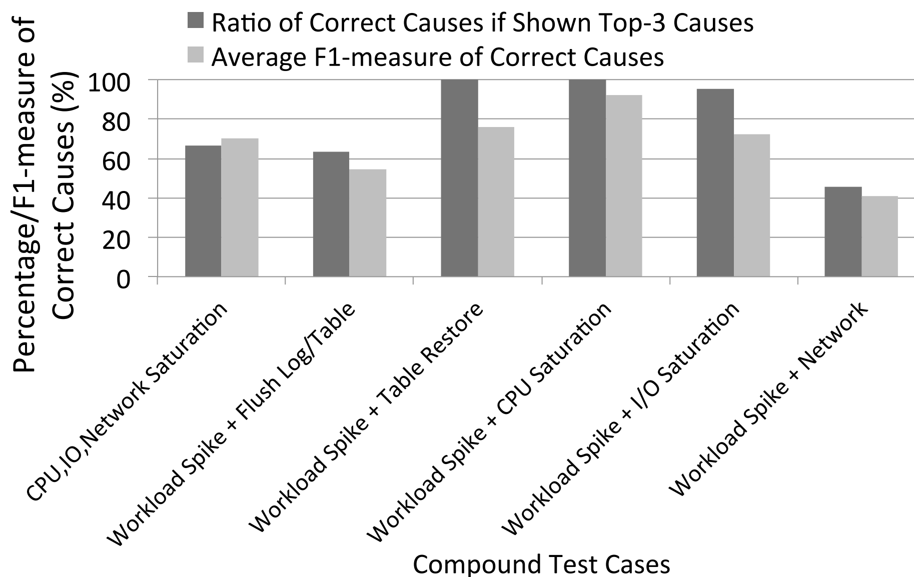


Figure 2.9: Ratio of the correct causes and their average F1-measure for compound situations (when top-3 causes are shown to the user).

Figure 2.9 demonstrates the ratio and the average F1-measure of correct causes (when three possible causes were offered to the user). On average, our explanation contained more than two-thirds of the correct causes, except for 'Workload Spike + Network Congestion'. For this dataset, `DBSherlock` missed the 'Workload Spike' and only returned 'Network Congestion' as the correct cause. This was because 'Network Congestion' had reduced the impact of 'Workload Spike' on the system (by slowing down the rate of incoming queries), and hence made it difficult for `DBSherlock` to identify their simultaneous occurrence.

2.8.8 Accuracy for other workloads

To confirm whether DBSherlock's capability of producing accurate explanations extends to other workloads besides TPC-C, we conducted additional experiments using the TPC-E benchmark [12]. Here, we used TPC-E with 3,000 customers, resulting in a 50GB data size. We generated the datasets and constructed merged causal models using a similar setup as in Section 2.8.5 (i.e., 5 datasets to construct merged causal models and 300 instances of explanations).

Type of Workload	Accuracy if shown top-1 cause	Accuracy if shown top-2 causes
TPC-C	98.0%	99.7%
TPC-E	92.5%	99.6%

Table 2.3: DBSherlock's accuracy for TPC-C and TPC-E workloads.

Table 2.3 compares the percentage of correct answers when the top-1 or top-2 causes are returned to the user for both TPC-C and TPC-E. When only the top-1 cause was shown to the user, the accuracy for the TPC-E workload slightly dropped to 92.5% on average. This was mainly due to DBSherlock's lower accuracy for 'Poor Physical Design' with TPC-E, as the effects of 'Poor Physical Design' and 'Lock Contention' on the system were not as significant as they were with TPC-C. The was due to the fact that TPC-E is much more read-intensive than TPC-C [77]. Nonetheless, DBSherlock still achieved an impressive accuracy of 99% on average with the TPC-E workload when the top-2 causes were shown to the user.

2.8.9 Over-fitting and Merged Causal Models

To verify if adding more datasets could further improve our merged models, we also ran a leave-one-out cross validation experiment. With 11 datasets for each case, we constructed causal models from 10 datasets and merged them. The final causal model then calculated confidence on the remaining dataset of each test case. Overall, the average confidence of the correct model increased slightly as shown in Figure 2.10a, but at the same time, the average margin of confidence decreased in some test cases as shown in Figure 2.10b.

The decrease in the average margin of confidence in some test cases suggests that merging more models than necessary can be ineffective. In other words, our proposed technique for merging causal models continues to widen the scope of relevant predicates while filtering irrelevant ones out. Once every irrelevant predicate has been filtered out, merging more models is not as effective. This is similar to the over-fitting phenomenon in machine learning. Nonetheless, DBSherlock still

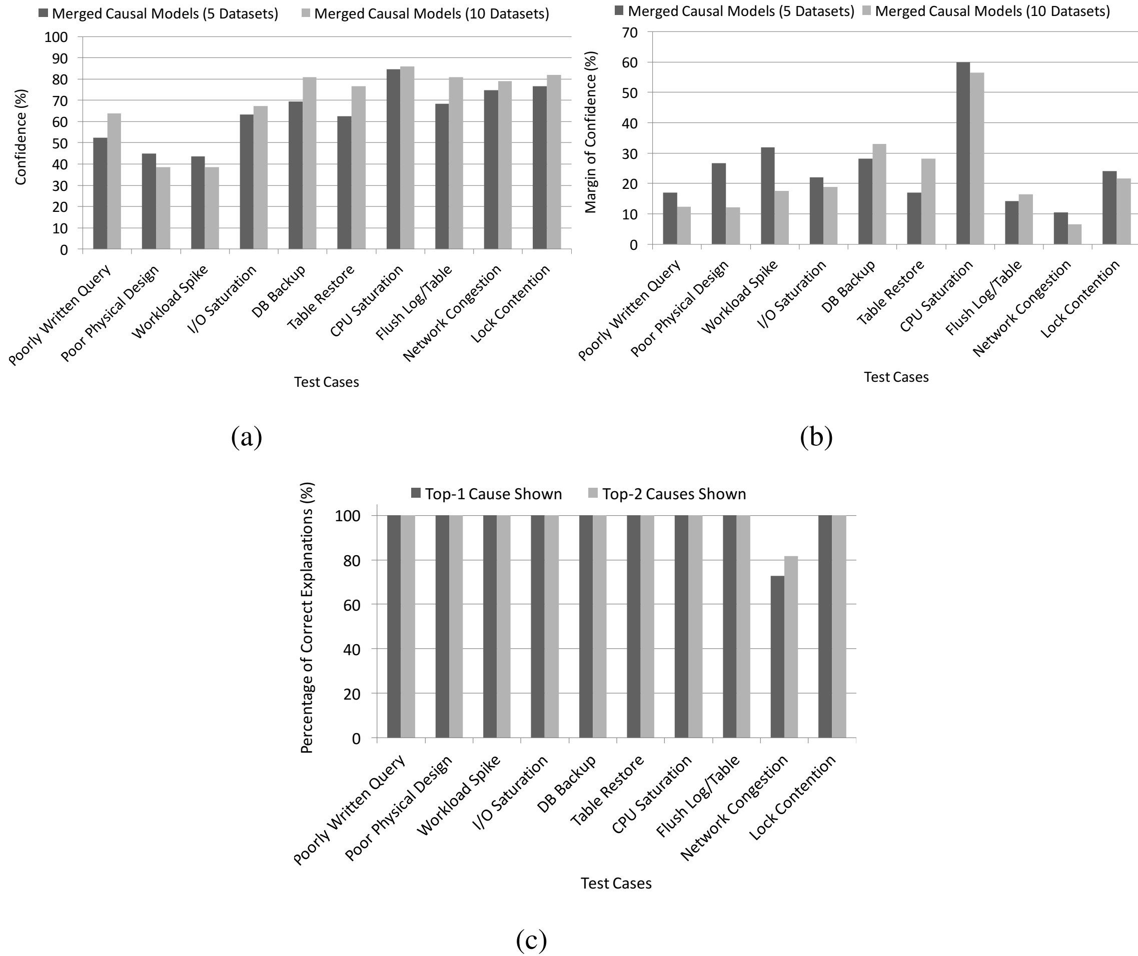


Figure 2.10: Evaluation of a merged causal model with 10 datasets in terms of (a) confidence, compared to a merged causal model with 5 datasets, (b) margin of confidence, compared to a merged causal model with 5 datasets, and (c) accuracy if top-k causes are returned.

succeeded in returning the correct cause among its top two explanations in every instance (except for 'Network Congestion'), as demonstrated in Figure 2.10c.

2.8.10 Rare Anomalies and Robustness Against Input Errors

As explained in Section 2.2.2, the user selects the abnormal regions manually after visual inspection of the performance plots (we have used the same method in our experiments—Section 2.8.2). However, users may not specify the regions with perfect precision. To understand how robust `DBSherlock` is to input errors caused by human mistakes, we conducted the following experiment. Using the same setup as Section 2.8.9, we extended the boundaries of the original anomaly region by 10% in one experiment and shortened them by 10% in another. We also ran a third

experiment where we randomly chose only two seconds of the original abnormal region as our input anomaly. The goal of this test case was to evaluate DBSherlock's effectiveness in diagnosing anomalies that are rare or only last a few seconds. For each dataset, we repeated each experiment 10 times and averaged the result.

Width of Abnormal Region	**Accuracy if shown top-1 cause**	**Accuracy if shown top-2 causes**
Original	94.6%	99.1%
10% Longer	95.5%	100%
10% Shorter	95.5%	97.3%
Two Seconds	74.6%	86.4%

Table 2.4: DBSherlock's robustness against rare and imperfect inputs.

Table 2.4 reports the percentage of correct answers when the top-1 or 2 causes are returned to the user. The accuracy did not change significantly when the span of the abnormal region was shorter or longer than the original one by 10%. More surprisingly, DBSherlock achieved reasonable accuracy even when the abnormal region was only two seconds long (e.g., the top-2 causes contained the correct explanation in 85% of the cases). These experiments show that DBSherlock remains effective even when the abnormal regions are not perfectly aligned with the actual anomaly or only last a very short period.

2.8.11 Different Parameters/Steps in DBSherlock

We conducted various experiments to study the effect of the individual steps and configurable parameters of our predicate generation algorithm on its accuracy.

To evaluate the different steps of our algorithm (from Section 2.4), we compared it against its simpler variants by omitting some of the steps each time. Since steps 1, 2 and 5 (i.e., *Creating a Partition Space*, *Partition Labeling* and *Extracting Predicates*) form the skeleton of our algorithm and cannot be excluded easily, we omitted the other two steps (i.e., *Partition Filtering* and *Filling the Gaps*). Table 2.5 reports the average margin of confidence and accuracy of each variant. Skipping either of the *Partition Filtering* or *Filling the Gaps* steps lowers the accuracy of our algorithm significantly (down to 0–10%). Without both, our algorithm fails to find any relevant predicates for explaining the given anomaly. This experiment underlines the significant contribution of these two steps towards our algorithm's overall accuracy.

Our predicate generation algorithm has three configurable parameters: the number of partitions (R), the anomaly distance multiplier (δ) and the normalized difference threshold (θ). We conducted experiments to study the effect of these parameters on the generated explanations. We ran our algorithm on every dataset with different values of each parameter and averaged its confidence,

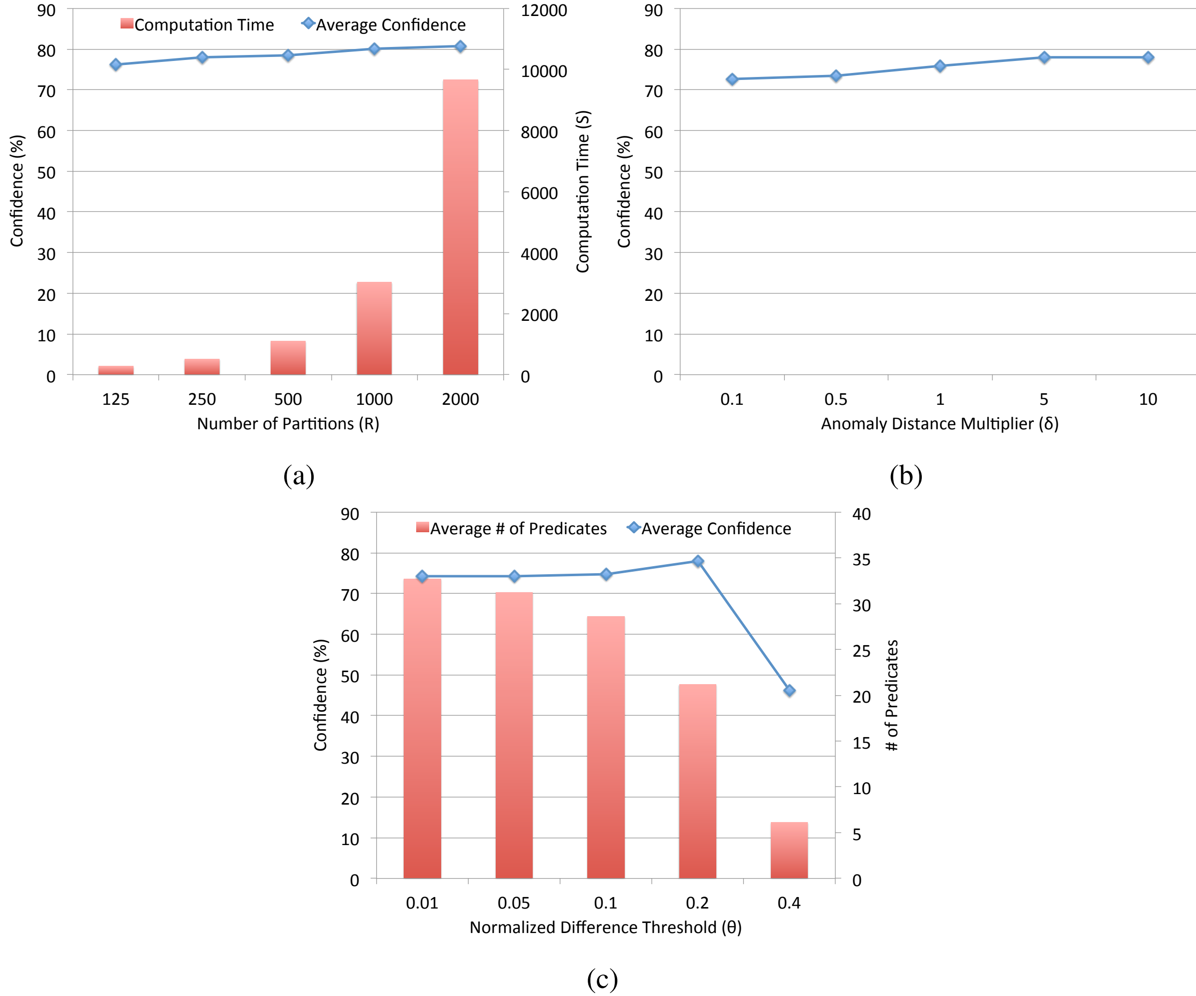


Figure 2.11: The effect of (a) the number of partitions on our algorithm's average confidence and computation time, (b) the anomaly distance multiplier on its confidence, and (c) the normalized difference threshold on its average confidence and number of generated predicates.

computation time, and number of generated predicates. For each experiment, we used 10 datasets to construct merged causal models and calculated their confidence on the remaining dataset. We used the default values of $\{R, \delta, \theta\}$ = $\{250, 10, 0.2\}$.

We varied R using the following values $\{125, 250, 500, 1000, 2000\}$. As shown in Figure 2.11a, R >1000 increased the computation time significantly, without much improvement in confidence. We also varied δ using the following values $\{0.1, 0.5, 1, 5, 10\}$. As shown in Figure 2.11b, and as expected, δ >1 favored more specific predicates and led to higher confidence.

Lastly, we varied the value of θ using the following values $\{0.01, 0.05, 0.1, 0.2, 0.4\}$. As shown in Figure 2.11c, increasing the value of θ reduced the number of generated predicates but increased their confidence slightly. However, the confidence dropped significantly with $\theta = 0.4$. This is because a large value of θ filters out most predicates, leaving only a few predicates that are

Algorithms	Overall avg. margin of confidence	Accuracy if shown top-1 cause
Original (all 5 steps)	37.4	94.6%
Without *Filling the Gaps*	9.3	10.1%
Without *Partition Filtering*	0.7	0%
Without *Filling the Gaps & Partition Filtering*	0	0%

Table 2.5: Contributions of the different steps of our predicate generation algorithm to the overall accuracy.

too specific to their training dataset and do not generalize to others.

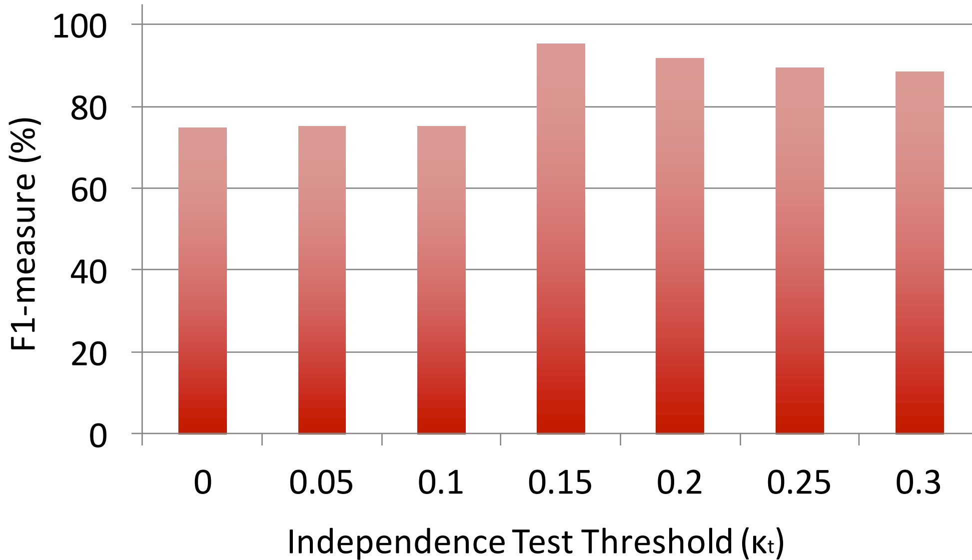


Figure 2.12: The effect of the parameter κ_t to average F1-measure.

We also performed a sensitivity analysis with the parameter κ_t that we use to filter secondary symptoms with the domain knowledge. We compared the average F1-measures with different values of κ_t. As demonstrated in Figure 2.12, the value of 0.15 for κ_t gave the highest average F1-measure.

2.8.12 DBSherlock's Accuracy with Automatic Anomaly Detection

We conducted an experiment to test `DBSherlock`'s accuracy when automatic anomaly detection is used and also compared against another anomaly detection algorithm, PerfAugur. For PerfAugur, we supplied the overall average latency as its performance indicator variable and used their naïve algorithm with the original scoring function to compute the abnormal region.

For this experiment, we have generated datasets with the longer duration (i.e., 10 minutes) of the normal workload to ensure that the normal region is larger than the abnormal region, necessary for automatic detection algorithm to distinguish between both. Then, causal models were constructed for each individual test case by merging the causal models from 10 datasets. Abnormal regions were manually specified with our 'ground-truth' knowledge of each test case, to simulate a perfect user diagnosing the problem. The anomaly detection algorithm described in Section 2.7 and PerfAugur then identified abnormal regions for the remaining dataset of each test case. This leave-one-out cross validation set-up is designed to test the effectiveness of DBSherlock using merged causal models, but in the absence of any user input. DBSherlock used the automatically detected region of anomaly as its input and generated explanations for each test case.

Detection Algorithms	**Accuracy if shown top-1 cause**	**Accuracy if shown top-2 causes**
Manual Anomaly Detection	94.6%	99.1%
Automatic Anomaly Detection	90%	95.5%
PerfAugur	77.3%	88.2%

Table 2.6: Ratio of the correct causes for different strategies.

As shown in Table 2.6, DBSherlock identified about 95% of the correct causes on average with our algorithm, when top-2 possible causes were shown to a user. Our anomaly detection algorithm also performed substantially better than PerfAugur's detection algorithm. This promising result demonstrates that DBSherlock can be used in an automated setting once it has enough user feedback for well-constructed causal models. An interesting future work is to integrate a domain-specific and a more sophisticated anomaly detection algorithm in DBSherlock.

2.8.13 User Study

We also performed a user study to evaluate the ability of our generated predicates in helping users diagnose the actual cause of performance problems. We asked various people who had some experience with databases to participate in a web-based questionnaire. We categorized the participants into three levels of competency based on their experience: *Preliminary DB Knowledge* (e.g., SQL knowledge or undergraduate course on databases), *DB Usage Experience*, and *DB Research or DBA Experience*. We used a few trivial questions to filter out spammers from genuine participants, leaving us with a total of 20 participants in our study. The questionnaire consisted of 10 multiple-choice questions. Each question had one correct cause and three randomly chosen incorrect causes. In each question, we presented a graph of average latency to our participants, with a pre-specified anomaly region and DBSherlock's generated predicates explaining the anomaly.

Background	# of participants	Avg. # of correct answers (out of 10)
Baseline (No Predicates)	N/A	2.5
Preliminary DB Knowledge	20	7.5
DB Usage Experience	15	7.8
DB Research or DBA Experience	13	7.8

Table 2.7: The summary of the user study.

Table 2.7 shows the summary of the user study. Here, the first row represents the random baseline, i.e., where no predicates are presented to the user. Participants with preliminary database knowledge were able to identify the correct cause in 75% of the cases. Participants with practical database usage or above identified the correct cause in 78% of the cases. This promising result shows that the predicates generated by `DBSherlock` can help the end user correctly diagnose anomalies in practice.

2.9 Related Work

Our work incorporates recent research in the fields of causality, performance diagnosis, and outlier detection.

Causality. Our work draws on the notion of causality proposed by Halpern and Pearl [129, 130]. We apply a simplified version of their causal model to introduce the notion of causality in our explanations. In the database literature, the notion of causality is brought together with data provenance [62, 80]. Meliou et al. [182] adapt the notion of causality to explain the cause of answers and non-answers in the output of database queries. In the context of probabilistic databases, Kanagal et al. [151] define the notion of an input tuple's *influence* on a query result. Scorpion [242] explains outliers in aggregate results of a query by unifying the concepts of *causality* and *influence*. Our notions of *normal* and *anomaly* are similar to Scorpion's *hold-out* and *outlier* sets, respectively.

Performance diagnosis. There have been many applications of performance diagnosis in databases [105], such as tuning query performance [47, 134], diagnosing databases that run on storage area networks [56], or parameter tuning [106]. Benoit [49] and Dias et al. [98] propose tools for automatic diagnosis of performance problems in commercial databases. However, [49] requires DBAs to provide a set of manual rules and [98] relies on detailed internal performance measurements from the DBMS (e.g., time spent in various modules of Oracle to process a query). More importantly, these tools lack explanatory features to answer 'why' a performance problem has occurred. In contrast, `DBSherlock` produces accurate explanations even without manual rules and using only aggregate statistics. Also, previous work has not accounted for the interaction of

the DBMS with the machine that it is running on. `DBSherlock` gives an explanation based on every statistic it can gather both inside and outside the DBMS.

In the context of MapReduce, there is work on automatic tuning of MapReduce programs [44, 144]. Here, the most relevant work is PerfXplain [155], which generates predicate-based explanations. PerfXplain helps debug MapReduce jobs, answering questions such as 'Why Job A is faster than Job B?'. `DBSherlock` is designed for OLTP workloads, its predicates are more accurate than those of PerfXplain (see Section 2.8.4), and can incorporate causal models.

There has been much work on automated performance diagnosis in other areas. Gmach et al. [116] use a *fuzzy controller* to remedy exceptional situations in an enterprise application. Their controller, however, requires the rules to be hard-coded and pre-defined by experts. In contrast, `DBSherlock` allows for causal models to be added, merged, and refined as new anomalies occur in the future. Further, while `DBSherlock` allows for incorporating domain knowledge, it can provide accurate explanations even without domain knowledge (see Section 2.8.6). Mahimkar et al. [178] propose a tool for troubleshooting IPTV networks, but assume that large correlation and regression coefficients among pairs of attributes imply causality. `DBSherlock` does not make this assumption.

Outlier detection. `DBSherlock` uses a simple but effective outlier detection strategy to autonomously monitor database performance (Section 2.7). Allowing users to choose from additional outlier detection algorithms (e.g., [64, 96, 154, 212, 213, 214, 226, 238, 239, 242, 249]) will make an interesting future work.

2.10 Summary

Performance diagnosis of database workloads is one of the most challenging tasks DBAs face on a daily basis. Besides basic visualization and logging mechanisms, current databases provide little help in automating this adhoc, tedious, and error-prone task. In this chapter, we presented `DBSherlock`, a framework that explains performance anomalies in the context of a complex OLTP environment. A user can select a region in a performance graph, which s/he thinks is abnormal, and ask `DBSherlock` to provide a diagnostic explanation for the observed anomaly. `DBSherlock` explains the anomaly in the form of predicates and possible causes produced by causal models. These explanations aid our users in diagnosing the correct cause of the performance problems more easily and more accurately. We also demonstrated that the confidence of our causal models can be increased via merging multiple causal models sharing the same cause. Our extensive experiments show that our algorithm is highly effective in identifying the correct explanations and is more accurate than the state-of-the art algorithm. As a much needed tool for coping with the increasing complexity of today's DBMS, `DBSherlock` is released as an open-source module in our workload management toolkit [4].

An important future work is to enable automatic actions for rectifying simple forms of performance anomaly (e.g., throttling certain tenants or triggering a migration), once they are detected and diagnosed with high confidence. We also plan to extend DBSherlock to go beyond creating causal models upon successful diagnoses, by documenting and storing the actions taken by the DBA to use as a suggestion for future occurrences of the same anomaly. Finally, DBSherlock's ideas might also be applicable to analytical workloads, e.g., in explaining performance problems caused by workload drifts [188].

CHAPTER 3

CliffGuard: A Principled Framework for Finding Robust Database Designs

3.1 Motivation

Modern databases come with designer tools (a.k.a. auto-tuning tools) that take certain parameters of a target workload (e.g., queries, data distribution, and various cost estimates) as input, and then use different heuristics to search the design space and find an optimal design (e.g., a set of indices or materialized views) within their time and storage budgets. However, these designs are only optimal for the input parameters provided to the designer. Unfortunately, in practice, these parameters are subject to many sources of uncertainty, such as noisy environments, approximation errors (e.g., in the query optimizer's cost or cardinality estimates [41]), and missing or time-varying parameters. Most notably, since future queries are unknown, these tools usually optimize for past queries in *hopes* that future ones will be similar.

Existing designer tools (e.g., Index Tuning Wizard [36] and Tuning Advisor in Microsoft SQL Server [73], Teradata's Index Wizard [59], IBM DB2's Design Advisor [259], Oracle's SQL Tuning Adviser [91], Vertica's DBD [163, 234], and Parinda for Postgres [179]) do not take into account the *influence of such uncertainties* on the optimality of their design, and therefore, produce designs that are *sub-optimal* and *remarkably brittle*. We call all these existing designers ***nominal***. That is, all these tools assume that their input parameters are precisely known and equal to some nominal values. As a result, overall performance often plummets as soon as future workload deviates from the past (say, due to the arrival of new data or a shift in day-to-day queries). These dramatic performance decays are severely disruptive for time-critical applications. They also waste critical human and computational resources, as dissatisfied customers request vendor inspections, often resulting in re-tuning/re-designing the database to restore the required level of performance.

Robust Designer— To overcome the shortcomings of nominal designers, we propose a new type of designers that are immune to parameter uncertainties as much as desired; that is, they are

robust. Our robust designer gives database administrators a *knob* to decide exactly how much nominal optimality to trade for a desired level of robustness. For instance, users may demand a set of optimal materialized views with an assurance that they must remain robust against change in their workload of up to 30%. A more conservative user may demand a higher degree of robustness, say 60%, at the expense of less nominal optimality. Robust designs are highly superior to nominal ones, as:

(a) Nominal designs are inherently brittle and subject to performance cliffs, while the performance of a robust design will degrade *more gracefully*.

(b) By taking uncertainties into account, robust designs can guard against worst-case scenarios, delivering a more consistent and predictable performance to time-sensitive applications.

(c) Given the highly non-linear and complex (and possibly non-convex) nature of database systems, a workload may have more than one optimal design. Thus, it is completely conceivable that a robust design may be nominally optimal as well (see [51,52] for such examples in other domains).

(d) A robust design can significantly reduce operational costs by requiring less frequent database re-designs.

Previous Approaches— There has been some pioneering work on incorporating parameter uncertainties in databases [41,70,82,103,181,206]. These techniques are specific to run-time query optimization and do not easily extend to physical designs. Other heuristics have been proposed for improving physical designs through workload compression (i.e., omitting workload details) [69,158] or modifying the query optimizer to return richer statistics [113]. Unfortunately, these approaches are not principled and thus do not necessarily guarantee robustness. (In Section 3.6.4, we compare against commercial databases that use such heuristics.)

To avoid these limitations, adaptive indexing schemes such as Database Cracking [128, 142] take the other extreme by completely ignoring the past workload in deciding which indices to build; instead of an *offline* design, they incrementally create and refine indices as queries arrive, *on demand*. However, even these techniques need to decide which subsets of columns to build an incremental index on.[1] Instead of completely relying on past workloads or abandoning the offline physical design, in this chapter we present a principled framework for directly maximizing robustness, which enables users to decide on the extent to which they want to rely on past information, and the extent of uncertainty they want to be robust against. (We discuss the merits of previous work in Section 3.7.)

[1]Moreover, on-demand and continuous physical re-organizations are not acceptable in many applications, which is why nearly all commercial databases still rely on their offline designers.

New Approach— Recent breakthroughs in Operations Research on robust optimization (RO) theory have created new hopes for achieving robustness and optimality in a principled and tractable fashion [51, 52, 78, 255]. In this chapter, we present the first attempt at applying RO theory to building a practical framework for solving one of the most fundamental problems in databases, namely finding the best physical design. In particular, we study the effects of workload changes on query latency. Since OLTP workloads tend to be more predictable (e.g., transactions are often instances of a few templates [185, 186]), we focus on OLAP workloads where exploratory and ad-hoc queries are quite common. Developing this robust framework is a departure from the traditional way of designing and tuning databases: from today's brittle designs to a principled world of robust designs that guarantee a predictable and consistent performance.

RO Theory— The field of RO has taken many strides over the past decade [51]. In particular, the seminal work of Bertsimas et al. [52] has been successfully applied to a number of drastically different domains, from nano-photonic design of telescopes [52] to thin-film manufacturing [55] and system-on-chip architectures [195]. To the best of our knowledge, developing a principled framework for applying RO theory to physical design problems is the first application of these techniques in a database context, which involves a number of unique challenges not previously faced in any of these other applications of RO theory.

A common misconception about the RO framework is that it requires knowledge of the extent of uncertainty, e.g., in our case, an upper bound on how much the future workload will deviate from the past one.[2] To the contrary, the power of the RO formulation is that it allows users to freely request any degree of robustness that they wish, say Γ, *purely* based on their own risk tolerance and preferences [48,53]. Regardless of whether the actual amount of uncertainty exceeds or stays lower than Γ, the RO framework guarantees will remain valid; that is, the delivered design is promised to remain optimal as long as the uncertainty remains below the user-requested threshold Γ, and beyond that (i.e., if uncertainty exceeds Γ) is in accordance to user's accepted degree of risk [53]. In other words, the beauty of RO theory is that it provides a framework for expressing and delivering reliability guarantees by decoupling them from the actual uncertainty in the environment (here, the future workload).

Contributions— In this chapter, we make these contributions:

- We introduce and summarize a principled algorithm, called `CliffGuard` [187], which adapts the state-of-the-art framework for solving non-convex RO problems. `CliffGuard`'s design is

[2]This misconception is caused by differing terminology used in other disciplines, such as mechanical engineering (ME) where "robust optimization" refers to a different type of optimization which requires some knowledge of the uncertainty of the physical environment [94]. The Operations Research notion of RO used in this chapter is called *reliability optimization* in the ME literature [230].

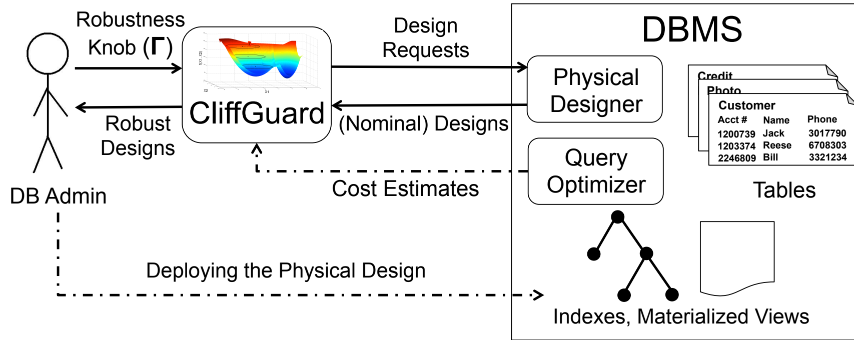


Figure 3.1: The `CliffGuard` architecture.

generic and can potentially work with any existing designers and databases without modifying their internals (Section 3.4).

- We implement and evaluate `CliffGuard` using two major commercial databases (HP Vertica and DBMS-X[3]) on two synthetic workloads as well as a real workload of 430+K OLAP queries issued by one of Vertica's major customers over a 1-year period (Section 3.6).

In summary, compared to Vertica's state-of-the-art commercial designer [163, 234], our robust designer reduces the average and maximum latency of queries on average by $7\times$ and $18\times$ (and up to $14\times$ and $40\times$), respectively. Similarly, `CliffGuard` improves over DBMS-X by 3–$5\times$.

3.2 System Overview

In this section, we provide an overview of `CliffGuard`.

Physical Database Designs— A physical design in a database is a set of auxiliary structures, often built *offline*, which are used to speed up future queries as they arrive. The type of auxiliary structures used often depend on the specific database architecture. Most databases use both materialized views and indices in their physical designs. Materialized views are typically more common in analytical workloads. Approximate databases use small samples of the data (rather than its entirety) to speed up query processing at the cost of accuracy [25, 26, 30, 66, 252, 253]. Physical designs in these systems consist of different types of samples (e.g., stratified on different columns [31, 66]). Some modern columnar databases, such as Vertica [234], build a number of column *projections*, each sorted differently. Instead of traditional indices, Vertica chooses a projection with the appropriate sort order (depending on the columns in the query) in order to locate relevant tuples quickly. In all these examples, the space of these auxiliary structures is extremely large if not infinite, e.g., there are $O(2^N \cdot N!)$ possible projections or indices for a table of N columns (i.e., different subsets

[3] DBMS-X is a major database system, which we cannot reveal due to the vendor's restrictions on publishing performance results.

and orders of columns). Thus, the physical design problem is choosing a small number of these structures using a fixed budget (in terms of time, space, or maintenance overhead) such that the overall performance is optimized for a target workload.

Design Principles— A major goal in the design of `CliffGuard` algorithm is compatibility with almost any existing database in order to facilitate its adoption in the commercial world. Thus, we have made two key decisions in our design. First, `CliffGuard` should operate alongside an existing (nominal) designer rather than replacing it. Despite their lack of robustness, existing designers are highly sophisticated tools hand-tuned over the years to find the best physical designs efficiently, given their input parameters. Because of this heavy investment, most vendors are reluctant to abandon these tools completely. However, some vendors have expressed interest in `CliffGuard` as long as it can operate alongside their existing designer and *improve* its output. Second, `CliffGuard` is designed to treat existing designers as a *black-box* (i.e., without modifying their internal implementations). This is to conform to the proprietary nature of commercial designers and also to widen the applicability of `CliffGuard` to different databases. By delegating the nominal designs to existing designers, `CliffGuard` remains a genetic framework agnostic to the specific details of the design objects (e.g., they can be materialized views, samples, indices, or projections).

These design principles have already allowed us to evaluate `Cliff- Guard` for two database products with drastically different design problems (i.e., Vertica and DBMS-X). Without requiring any changes to their internal implementations, `CliffGuard` significantly improves on the sophisticated designers of these leading databases (see Section 3.6). Thus, we believe that `CliffGuard` can be easily used to speed up other database systems as well.

Architecture— Figure 3.1 depicts the high-level workflow of how `CliffGuard` is to be used alongside a database system. The database administrator states her desired degree of robustness Γ to `CliffGuard`, which is located outside the DBMS. `CliffGuard` in turn invokes the existing physical designer via its public API. After evaluating the output (nominal) design sent back from the existing designer, `CliffGuard` may decide to manipulate the existing designer's output by merely modifying some of its input parameters (in a principled manner) and invoking its API again. `CliffGuard` repeats this process, until it is satisfied with the robustness of the design produced by the nominal designer. The final (robust) design is then sent back to the administrator, who may decide to deploy it in the DBMS.

3.3 Problem Formulation

In this section, we present a formulation of robustness used by `CliffGuard` in the context of physical database design. This formulation allows us to employ recently proposed ideas in the theory

of robust optimization (RO) and develop a principled and effective algorithm for finding robust database designs, which will be described in Section 3.4. First, we define some notations.

Notations— For a given database, the **design space** $\mathcal{S}$ is the set of all possible structures of interest, such as indices on different subsets of columns, materialized views, different samples of the data, or a combination of these. For example, Vertica's designer [234] materializes a number of *projections*, each sorted differently:

```
CREATE PROJECTION   projection_name
  AS SELECT   col1, col2, ..., colN
  FROM   anchor_table
  ORDER BY   col1', col2', ..., colK';
```

Here, $\mathcal{S}$ is extremely large due to the exponential number of possible projections. Similarly, for decisions about building materialized views or (secondary) indices, $\mathcal{S}$ will contain all such possible structures. Existing database designers solve the following optimization problem (or aim to[4]):

$$D^{nom} = \mathbb{D}(W_0, B) = \underset{D \subseteq \mathcal{S},\ price(D) \leq B}{\textit{ArgMin}} \ f(W_0, D) \tag{3.1}$$

where W_0 is the target **workload** (e.g., the set of user queries), B is a given **budget** (in terms of storage or maintenance overhead), $\mathbb{D}$ is a **nominal designer** that takes a workload and budget as input parameters, $price(D)$ is the **price** of choosing D (e.g., the total size of the projections in D), and $f(W_0, D)$ is our **cost function** for executing workload W_0 using design D (e.g., f can be the query latency). We call such designs **nominal** as they are optimal for the nominal value of the given parameters (e.g., the target workload). All existing designers [36, 91, 179, 234, 259] are *nominal*: they either minimize the expression above directly, or follow other heuristics aimed at approximate minimization. Despite several heuristics to avoid over-fitting a given workload (e.g., omitting query details [69, 158]), nominal designers suffer from many shortcomings in practice; see Sections 3.1 and 3.6.4.

Robust Designs— `CliffGuard`'s goal is finding designs that are robust against *worst-case* scenarios that can arise from uncertain situations. This concept of *robustness* can be illustrated using the toy example of Figure 3.2, which features a design space with only three possible designs and a toy workload that is represented by a single real-value parameter μ. When our current estimate of μ is μ_0, a nominal designer will pick design D_1 since it minimizes the cost at μ_0. But if we want a design that remains optimal even if our parameter changes by up to Γ, then a robust designer will pick design D_2 instead of D_1, even though the latter has a lower cost at μ_0. This is because the

[4]Existing designers often use heuristics or greedy strategies [180], which lead to *approximations* of the nominal optima.

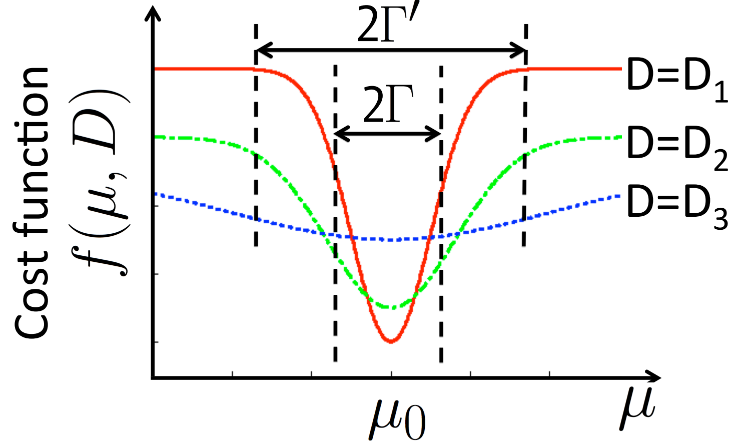


Figure 3.2: Design D_1 is nominally optimal at μ_0 while designs D_2 and D_3 are robust against an uncertainty of $\pm\Gamma$ and $\pm\Gamma'$ in our parameter μ_0, respectively.

worst-case cost of D_2 over the $[\mu_0 - \Gamma, \mu_0 + \Gamma]$ is lower than that of D_1; that is, D_2 is robust against uncertainty of up to Γ. Similarly, if we decide to guard against a still greater degree of uncertainty, say for an estimation error as high as $\Gamma' > \Gamma$, a robust designer would this time pick D_3 instead of D_2, as the former has a lower worst-case cost in $[\mu_0 - \Gamma', \mu_0 + \Gamma']$ than the other designs.

Formally, a robust design D^{rob} can be defined as:

$$D^{rob} = \tilde{\mathbb{D}}(W_0, B, \mathcal{U}) = \underset{D \subseteq \mathcal{S},\ price(D) \leq B}{ArgMin} \ \underset{W \in \mathcal{U}(W_0)}{Max} \ f(W, D)$$

where $\mathcal{U}(W_0)$ defines an uncertainty region around our target workload W_0. Here, $\tilde{\mathbb{D}}$ is a robust designer that will search for a design that minimizes the cost function regardless of where the target workload lands in this uncertainty region. In other words, this MiniMax formulation of robustness defines a robust design as one with the **best worst-case performance**.

Although the uncertainty region $\mathcal{U}(W_0)$ does not have to be circular, for ease of presentation in this chapter, we always define $\mathcal{U}(W_0)$ as a circular region of radius $\Gamma \geq 0$ centered at W_0, which we call the Γ**-neighborhood** of W_0. For instance, the Γ-neighborhood will be an interval when $W_0 \in \mathbb{R}$ (see Figure 3.2) and a circle when $\Gamma \in \mathbb{R}^2$. Since database workloads are not easily represented as real numbers, we need to use a distance function to define the Γ-neighborhood of a database workload W_0, namely:

$$D^{rob} = \tilde{\mathbb{D}}(W_0, B, \Gamma) = \underset{D \subseteq \mathcal{S},\ price(D) \leq B}{ArgMin} \ \underset{\delta(W, W_0) \leq \Gamma}{Max} \ f(W, D) \tag{3.2}$$

Here, $\delta(.)$ is a user-defined distance function that takes a pair of workloads and returns a non-negative real number as their distance. Formulation (3.2) allows users to express their desired level of robustness by choosing the value of $\Gamma \geq 0$, where the larger the Γ, the more robust their design

is. Note that a nominal design is a special case of a robust design where $\Gamma = 0$. In the rest of this chapter, we will not explicitly mention the $price(D) \leq B$ constraint in our notations, but it will always be implied in both nominal and robust designs.

A Knob for Robustness— As mentioned in Section 3.1, the role of Γ in RO formulation (3.2) is sometimes misunderstood to be an upper bound on the degree of uncertainty, i.e., Γ should be chosen such that the future workload W will lie in W_0's Γ-neighborhood. To the contrary, the beauty of formulation (3.2) is that it allows users to choose any Γ value based purely on their own business needs and risk tolerance, regardless of the actual amount of uncertainty in the future. In other words, Γ is *not* an upper bound on the actual uncertainty in the environment, but rather the amount of actual uncertainty that the user decides to guard against. This is a subtle but important distinction, because robustness comes at the price of reduced nominal optimality. In the example of Figure 3.2, D_2 is robust against a greater degree of uncertainty than D_2 but is nominally more expensive at $\mu = \mu_0$. Therefore, it is important to interpret Γ as a *robustness knob* and not a prediction of future uncertainty.

The choice of Γ depends completely on the end users' risk tolerance and is not the focus of this chapter. Our framework will deliver a design that guarantees the requested level of robustness for any value of Γ chosen by the user. For instance, a user may take the simplest approach and use the sequence of workload changes over the past N windows of queries, say

$$\delta(W_0, W_1), \delta(W_1, W_2), \cdots, \delta(W_{N-1}, W_N)$$

and take their average, max, or $k\times$max (for some constant $k>1$) as a reasonable choice of Γ when finding a robust design for W_{N+1} using W_N. Alternatively, a user may employ more sophisticated techniques (e.g., timeseries forecasting [65]) to obtain a more accurate prediction for $\delta(W_N, W_N + 1)$. Regardless of the strategy, the actual uncertainty can always exceed a user's predictions. However, this problem is no different from any other provisioning problem. For instance, many customers provision their database resources according to, say, $3\times$ their current peak load. This means that according to their business needs, they accept the risk of their future workload suddenly increasing by $4\times$. This is analogous to the user's choice of Γ here. Also, note that even if users magically knew the exact value of $\delta(W_N, W_{N+1})$ in advance, the existing nominal designers' performance would remain the same since they have no mechanism for incorporating a bounded uncertainty into their analysis. (A nominal designer would only perform better if we knew the actual W_{N+1} and not just its distance from W_N.) As previously explained, while our proposed designer does not *require* any prior knowledge of the uncertainty in order to deliver the user's robustness requirements, it can naturally incorporate additional of the future workload if made available by the user.

In Section 3.6.5, we study the effects of different Γ choices and show that, in practice, our algorithm performs no worse than the nominal designer even when presented with poor (i.e., extremely low or extremely high) Γ choices.

3.4 Summary of CliffGuard's Algorithm

In the previous section, we provided the RO formulation of the physical design for databases. Over the past decade, there have been many advances in the theory of RO for solving problems with similar formulations as (3.2), for many different classes of cost functions and uncertainty sets (see [51] for a recent survey). Here, the most relevant to our problem is the seminal work of Bertsimas et al. [52], hereon referred to as the BNT algorithm. Unlike most RO algorithms, BNT does not require the cost function to have a closed-form. This makes BNT an ideal match for database context: the cost function is often the query latency, which does not have an explicit closed-form, i.e., latency can only be *measured* by executing the query itself or *approximated* using the query optimizer's cost estimates. BNT's second strength is that it does not require convexity: BNT guarantees a global robust solution when the cost function is convex, and convergence to a local robust solution even when it is not convex. Given the complex nature of modern databases, establishing convexity for query latencies can be difficult (e.g., in some situations, additional load can reduce latency by improving the cache hit rate [185]).[5]

First, we provide background on the BNT framework in Section 3.4.1. Then, in Section 3.4.2, we summarize the BNT-based `CliffGuard` algorithm. For more details on the algorithm, please refer to [189].

3.4.1 The BNT Algorithm

In this section, we offer a geometric interpretation of the BNT algorithm for an easier understanding of the main ideas behind the algorithm. (Interested readers can find a more formal discussion of the algorithm in the Operations Research literature [52].)

We use Figure 3.3a to illustrate how the BNT algorithm works. Here, imagine a simplified world in which each decision is a 2-dimensional point in Euclidean space. Since the environment is noisy or unpredictable, the user demands a decision x^* that comes with some *reliability* guarantees. For example, instead of asking for a decision x^* that simply minimizes $f(x)$, the user requires an

[5] When the cost function is non-convex, the output of existing nominal designers is also only *locally* optimal. Thus, even in these cases, finding a local robust optimum is still a worthwhile endeavor.

x^* whose worst-case cost is minimized for arbitrary noise vectors Δx within a radius of Γ, namely:

$$x^* = \underset{x}{ArgMin} \underset{||\Delta x||_2 \leq \Gamma}{Max} \quad f(x + \Delta x) \tag{3.3}$$

Here, $||\Delta x||_2$ is the length (L_2-norm) of the noise vectors. This means that the user expresses his/her reliability requirement as an uncertainty region, here a circle of radius Γ, and demands that our $f(x^* + \Delta x)$ still be minimized no matter where the noisy environment moves our initial decision within this region. This uncertainty region (i.e., the Γ-neighborhood) is shown as a shaded disc in Figure 3.3a.

To meet the user's reliability requirement, the BNT algorithm takes a starting point, say $\hat{x}$, and performs a number of iterations as follows. In each iteration, BNT first identifies all the points within the Γ-neighborhood of $\hat{x}$ that have the highest cost, called the *worst-neighbors* of $\hat{x}$. In Figure 3.3a, there are four worst-neighbors, shown as $u_1, \cdots, u_4$. Let $\Delta x_1, \cdots, \Delta x_4$ be the vectors that connect $\hat{x}$ to each of these worst-neighbors, namely $u_i = \hat{x} + \Delta x_i$ for i=$1, 2, 3, 4$.

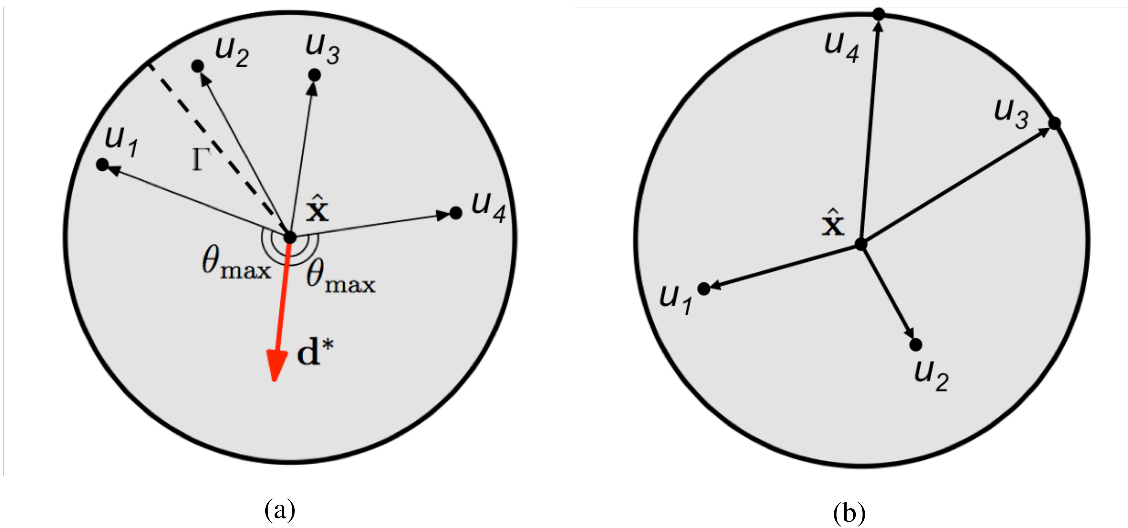


Figure 3.3: (a) A descent direction d^* is one that moves away from *all* the worst-neighbors ($\theta_{max} \geq 90°$); (b) here, due to the location of the worst-neighbors, no descent direction exists.

Once the worst-neighbors of $\hat{x}$ are identified, the BNT algorithm finds a direction that moves away from all of them. This direction is called the *descent direction*. In our geometric interpretation, a descent direction $\vec{\mathbf{d}}^*$ is one that maximizes the angle θ in Figure 3.3a by halving the reflex angle between the vectors connecting $\hat{x}$ to u_1 and u_4. The BNT algorithm then takes a small step along this descent direction to reach a new decision point, say $\hat{x}'$, which will be at a greater distance from all of the worst-neighbors of $\hat{x}$. The algorithm repeats this process by looking for the new worst-neighbors in the Γ-neighborhood of $\hat{x}'$. (Bertsimas et al. prove that taking an appropriately-sized step along the descent direction reduces the worst-case cost at each iteration [52].) The

algorithm ends (i.e., a robust solution is found) when no descent direction can be found. Figure 3.3b illustrates this situation, as any direction of movement within the Γ-neighborhood will bring the solution closer to at least one of the worst-neighbors. (Bertsimas et al. prove that this situation can only happen when we reach a local robust minimum, which will also be a global robust minimum when the cost function is convex.)

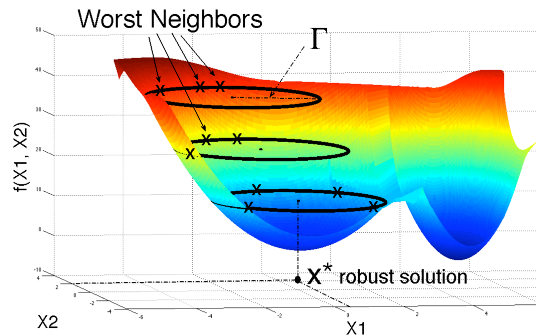


Figure 3.4: Geometric interpretation of the iterations in BNT.

To visually demonstrate this convergence, we again use a geometric interpretation of the algorithm, as depicted in Figure 3.4. In this figure, the decision space consists of two-dimensional real vectors $(x_1, x_2) \in \mathbb{R}^2$ and the $f(x_1, x_2)$ surface corresponds to the cost of different points in this decision space. Here, the Γ-neighborhood in each iteration of BNT is shown as a transparent disc of radius Γ. Geometrically, to move away from the worst-neighbors at each step is equivalent to sliding down this disc along the steepest direction such that the disc always remains within the cost surface and parallel to the (x_1, x_2) plane. The algorithm ends when this disc's boundary touches the cost surface and cannot be sliced down any further without breaking through the cost surface—this is the case with the bottom-most disc in Figure 3.4; when this condition is met, the center of this disc represents a locally robust solution of the problem (marked as x^*) and its worst-neighbors lie on the boundary of its disc (marked as $\times$). The goal of BNT is to quickly find such discs and converge to the locally robust optimum.

The pseudocode of BNT is presented in Algorithm 2. Here, x_k is the current decision at the k'th iteration. As explained above, each iteration consists of two main steps: finding the worst-neighbors (neighborhood exploration, Line 5) and moving away from those neighbors if possible (local robust move, Lines 7–16).

Theoretical Guarantees— When $f(x)$ is continuously differentiable with a bounded set of minimum points, Bertsimas et al. [52] show that their algorithm converges to the local optimum of the robust optimization problem (3.3), as long as the steps sizes t_k (Line 14 in Algorithm 2) are chosen such that $t_k > 0$, $\lim_{k\to\infty} t_k = 0$, and $\sum_{k=1}^{\infty} t_k = \infty$. For convex cost surfaces, this solution is also the global optimum. (With non-convex surfaces, BNT needs to be repeated from different starting

points to find multiple local optima and choose one that is more globally optimal.[6])

Inputs: Γ: the radius of the uncertainty region,
$f(x)$: the cost of design x
Output: x^*: a robust design, i.e., $x^* = \underset{x}{ArgMin} \underset{||\Delta x||_2 \leq \Gamma}{Max} f(x + \Delta x)$

1 $x_1 \leftarrow$ pick an arbitrary vector **// the initial decision**
2 $k \leftarrow 1$ **//** k **is the number of iterations so far**
3 **while** *true* **do**
    // Neighborhood Exploration:
5     $U \leftarrow$ Find the set of worst-neighbors of x_k within its Γ-neighborhood
    // Robust Local Move:
7     $\vec{\mathbf{d}}^* \leftarrow$ FindDescentDirection(x_k, U)
    // See Fig 3.3a for FindDescentDirection**'s geometric intuition (formally defined in [189])**
9   **if** there is no such direction $\vec{\mathbf{d}}^*$ pointing away from all $u \in U$ **then**
11         $x^* \leftarrow x_k$ **// found a local robust solution**
12   **return** x^*
    else
14         $t_k \leftarrow$ choose an appropriate step size
15         $x_{k+1} \leftarrow x_k + t_k \cdot \vec{\mathbf{d}}^*$ **// move along the descent direction**
16         $k \leftarrow k + 1$ **// go to next iteration**

Algorithm 2: Generic robust optimization via gradient descent.

3.4.2 CliffGuard: Algorithm

In this section, we describe the algorithm of `CliffGuard`, which builds upon BNT's principled framework by tailoring it to the problem of physical database design.

Before presenting this algorithm, we need to clarify a few notional differences. Unlike BNT, where the cost function $f(x)$ takes a single parameter x, the cost in `CliffGuard` is denoted as a two-parameter function $f(W, D)$ where W is a given workload and D is a given physical design. In other words, each point x in this space is a pair of elements (W, D). However, unlike BNT where vector x can be updated in its entirety, in `CliffGuard` (or any database designer) we only update the design element D; this is because the database designer can propose a new physical

[6]When $f(x)$ is non-convex, the output of existing designers is also a local optimum. Thus, even in this case, finding local robust optima is still preferable (to a local nominal optimum).

design to the user, but cannot impose a new workload on her as a means to improve robustness.

Algorithm 3 presents the pseudocode for `CliffGuard`. Like Algorithm 2, Algorithm 3 iteratively explores a neighborhood to find the worst-neighbors, then moves farther away from these neighbors in each iteration using an appropriate direction and step size. However, to apply these ideas in a database context, Algorithm 3 differs from Algorithm 2 in the following important ways.

Initialization (Algorithm 3, Lines 1–2)— `CliffGuard` starts by invoking the existing designer $\mathbb{D}$ to find a nominal design D for the initial workload W_0. (Later, D will be repeatedly replaced by designs that are more robust.) `CliffGuard` also creates a finite set of perturbed workloads $P = \{W_1, \cdots, W_n\}$ by sampling the workload space in the Γ-neighborhood of W_0. In other words, given a distance metric δ, we find n workloads $W_1, \cdots, W_n$ such that $\delta(W_i, W_0) \leq \Gamma$ for $i = 1, 2, \cdots, n$. ([189] discusses in more details about how to define δ for database workloads, how to choose n, and how to sample the workload space efficiently.) Next, as in BNT, `CliffGuard` starts an iterative search with a neighborhood exploration and a robust local move in each iteration.

Neighborhood Exploration (Algorithm 3, Line 6)— To find the worst-neighbors, in `CliffGuard` we need to also take the current design D into account (i.e., the set of worst-case neighbors of W_0 will depend on the physical design that we choose). Given that we cannot rely on the differentiability (or even continuity) of worst-case cost function, we use the worst-case costs on sampled workloads P a proxy; instead of solving

$$\underset{\delta(W,W_0)\leq\Gamma}{Max} f(W, D) \tag{3.4}$$

we solve

$$\underset{W\in P}{Max} f(W, D) \tag{3.5}$$

Note that (3.5) cannot provide an unbiased approximation for (3.4) simply because P is a finite sample, and finite samples lead to biased estimates for extreme statistics such as min and max [233]. Thus, we do not rely on the nominal value of (3.5) to evaluate the quality of a design. Rather, `CliffGuard` uses the solutions to (3.5) as a proxy to guide the search in moving away from highly (though not necessarily the most) expensive neighbors. In the actual implementation, `CliffGuard` further mitigate this sampling bias by loosening its selection criterion to include all neighbors that have a high-enough cost (e.g., top-K or top 20%) instead of only those that have the maximum cost. To implement this step, we simply enumerate each workload in P and measure its latency on the given design.

Robust Local Move (Algorithm 3, Lines 8–15)— To find equivalent database notions for finding and moving along a descent direction (C3 and C4), we use the following idea. The ultimate goal of finding and moving along a descent direction is to reduce the worst-case cost of the current design.

In CliffGuard, we can achieve this goal directly by *manipulating* the existing designer by feeding it a mixture of the existing workload and its worst-neighbors as a single workload.[7] The intuition is that since nominal designers (by definition) produce designs that minimize the cost of their input workload, the cost of its previous worst-neighbors will no longer be as high, which is equivalent to moving its design farther away from those worst-neighbors. The questions then are (i) how do we mix these workloads, and (ii) what if the designer's output leads to a higher worst-case cost?

The answer to question (i) is a weighted union, where we take the union of all the queries in the original workload as well as those in the worst-neighbors, after weighting the latter queries according to a scaling factor α, their individual frequencies of occurrence in their workload, and their latencies against the current design. Taking latencies and frequencies into account encourages the nominal designer to seek designs that reduce the cost of more expensive and/or popular queries. Scaling factor α, which serves the same purpose as step-size in BNT, allows CliffGuard to control the distance of movement away from the worst-neighbors.

We also need to address question (ii) because unlike BNT, where the step size t_k could be computed to ensure a reduction in the worst cost, here α factor may in fact lead to a worse design (e.g., by moving too far from the original workload). To solve this problem, CliffGuard dynamically adjusts the step-size using a common technique called *backtracking line search* [57], similar to a binary-search. Each time the algorithm succeeds in moving away from the worst-neighbors, we consider a larger step size (by a factor $\lambda_{success} >1$) to speed up the search towards the robust solution, and each time we fail, we reduce the step size (by a factor $0< \lambda_{failure} <1$) as we may have moved past the robust solution (hence observing a higher worst-case cost).

Termination (Algorithm 3, Lines 16–19)— We repeat this process until we find a local robust optimum (or reach the maximum number of steps, when under a time constraint).

[7] Remember that existing designers only take a single workload as their input parameter.

Inputs: Γ: the desired degree of robustness,
δ: a distance metric defined over pairs of workloads,
W_0: initial workload,
$\mathbb{D}$: an existing (nominal) designer,
f: the cost function (or its estimate),
Output: D^*: a robust design, i.e., $D^* = \underset{D}{ArgMin} \underset{\delta(W-W_0)\leq\Gamma}{Max} f(W, D)$

1 $D \leftarrow \mathbb{D}(W_0)$ **// Invoke the existing designer to find a nominal design for** W_0
2 $P \leftarrow \{W_i \mid 1 \leq i \leq n, \ \delta(W_i, W) \leq \Gamma\}$ **// Sample some perturbed workloads in the Γ-neighbor of** W_0
3 Pick some $\alpha > 0$ **// some initial size for the descending steps**
4 **while** *true* **do**
 // Neighborhood Exploration:
6 $U \leftarrow \{\tilde{W}_1, \cdots, \tilde{W}_m\}$ where $\tilde{W}_i \in P$ and $f(\tilde{W}_i, D) = \underset{W\in P}{Max} f(W, D)$ **// Pick perturbed workloads with the worst performance on** D
 // Robust Local Move:
8 $W_{moved} \leftarrow$ MoveWorkload($W_0, \{\tilde{W}_1, \cdots, \tilde{W}_m\}, f, D, \alpha$) **//Build a new workload by moving closer to W_0's worst-neighbors (see Alg. 4)**
9 $D' \leftarrow \mathbb{D}(W_{moved})$ **// consider the nominal design for W_{moved} as an alternative design**
10 **if** $\underset{W\in P}{Max} f(W, D') < \underset{W\in P}{Max} f(W, D)$ ***// Does D' improve on the existing design in terms of the worst-case performance?***
 then
12 $D \leftarrow D'$ **// Take D' as your new design**
13 $\alpha \leftarrow \alpha * \lambda_{success}$ (for some $\lambda_{success} > 1$) **// increase the step size for the next move along the descent direction**
 else
15 $\alpha \leftarrow \alpha * \lambda_{failure}$ (for some $\lambda_{failure} < 1$) **// consider a smaller step next time**
16 **if** your time budget is exhausted or many iterations have gone with no improvements
 then
 $D^* \leftarrow D$ **// the current design is robust**
19 **return** D^*

Algorithm 3: The `CliffGuard` algorithm.

3.5 Expressing Robustness Guarantees

In this section, we describe a database-specific distance metric δ in `CliffGuard` so that users can express their robustness requirements by specifying a Γ-neighborhood (as an uncertainty set, described in Section 3.3) around a given workload W_0, and demanding that their design must be robust for any future workload W as long as $\delta(W_0, W) \leq \Gamma$. Thus, users can demand arbitrary degrees of robustness according to their performance requirements. For mission-critical applications

more sensitive to sudden performance drops, users can be more conservative (specifying a larger Γ). At the other extreme, users expecting no change (or less sensitive to it) can fall back to the nominal case ($\Gamma = 0$).

A distance metric δ must satisfy the following criteria to be effectively used in our BNT-based framework (the intuition behind these requirements can be found in the technical report [189]):

1. *Soundness*, which requires that the smaller the distance $\delta(W_1, W_2)$, the better the performance of W_2 on W_1's nominally optimal design. Formally, we call a distance metric *sound* if it satisfies:

$$\delta(W_1, W_2) \leq \delta(W_1, W_3) \Rightarrow f(W_2, \mathbb{D}(W_1)) \leq f(W_3, \mathbb{D}(W_1)) \tag{3.6}$$

2. δ should account for intra-query similarities; that is, if $r_i^1 > r_i^2$ and $r_j^1 < r_j^2$, the distance $\delta(W_1, W_2)$ should become smaller based on the similarity of the queries q_i and q_j, assuming the same frequencies for the other queries.

3. δ should be symmetric; that is, $\delta(W_1, W_2) = \delta(W_2, W_1)$ for any W_1 and W_2. (This is needed for the theoretical guarantees of the BNT framework.)

4. δ must satisfy the *triangular property*; that is, $\delta(W_1, W_2) \leq \delta(W_1, W_3) + \delta(W_3, W_2)$ for any W_1, W_2, W_3. (This is an implicit assumption in almost all gradient-based optimization techniques, including BNT.)

Before introducing a distance metric fulfilling these criteria, we need to introduce some notations. Let us represent each query as the union of all the columns that appear in it (e.g., unioning all the columns in the `select`, `where`, `group by`, and `order by` clauses). With this oversimplification, two queries will be considered identical as long as they reference the same set of columns, even if their SQL expressions, query plans, or latencies are substantially different. Using this representation, there will be only $2^n - 1$ possible queries where n is the total number of columns in the database (including all the tables). (Here, we ignore queries that do not reference any columns.) Thus, we can represent a workload W with a $(2^n - 1)$-dimensional vector $V_W = \langle r_1, \cdots, r_{2^n-1} \rangle$ where r_i represents the normalized frequency of queries that are represented by the i'th subset of the columns for $i = 1, \cdots, 2^n - 1$. With this notation, we can now introduce our Euclidean distance for database workloads as:

$$\delta_{euclidean}(W_1, W_2) = |V_{W_1} - V_{W_2}| \times S \times |V_{W_1} - V_{W_2}|^T \tag{3.7}$$

Here, S is a $(2^n - 1) \times (2^n - 1)$ similarity matrix, and thus $\delta_{euclidean}$ is always a real-valued number (i.e., 1×1 matrix). Each $S_{i,j}$ entry is defined as the total number of columns that are present only

in q_i or q_j (but not in both), divided by $2 \cdot n$. In other words, $S_{i,j}$ is the Hamming distance between the binary representations of i and j, divided by $2 \cdot n$. Hamming distances are divided by $2 \cdot n$ to ensure a normalized distance, i.e., $0 \leq \delta_{euclidean}(W_1, W_2) \leq 1$.

One can easily verify that $\delta_{euclidean}$ satisfies criteria (b), (c), and (d). In Section 3.6.3, we empirically show that this distance metric also satisfies criterion (a) quite well. Finally, even though V_W is exponential in the number of columns n, it is merely a conceptual model; since V_W is an extremely sparse matrix, most of the computation in (3.7) can be avoided. In fact, $\delta_{euclidean}$ can be computed in $O(T^2 \cdot n)$ time and memory complexity, where T is the number of input queries (e.g., in a given query log).

Limitations— $\delta_{euclidean}$ has a few limitations. First, it does not factor in the clause in which a column appears. For instance, for fast filtering, it is more important for a materialized view to cover a column appearing in the `where` clause than one appearing only in the *select* clause. This limitation, however, can be easily resolved by representing each query as a 4-tuple $\langle v_1, v_2, v_3, v_4 \rangle$ where v_1 is the set of columns in the `select` clause and so on. We refer to this distance as $\delta_{separate}$, as we keep columns appearing in different clauses separate. $\delta_{separate}$ differs from $\delta_{euclidean}$ only in that it creates 4-tuple vectors, but it is still computed using Equation (3.7).

The second (and more important) limitation is that $\delta_{euclidean}$ may ignore important aspects of the SQL expression if they do not change the column sets. For example, presence of a `join` operator or using a different query plan can heavily impact the execution time, but are not captured by $\delta_{euclidean}$. In fact, as a stricter version of requirement (3.6), a better distance metric will be one that for all workloads W_1, W_2, W_3 and *arbitrary* design D satisfies:

$$\begin{aligned} \delta(W_1, W_2) \leq \delta(W_1, W_3) \quad &\Rightarrow \\ |f(W_2, D) - f(W_1, D)| \quad &\leq \quad |f(W_3, D) - f(W_1, D)| \end{aligned} \tag{3.8}$$

In other words, the distance functions should directly match the performance characteristics of the workloads (the lower their distance, the more similar their performance).

First, requirement (3.9) is unnecessary for our purposes. `CliffGuard` only relies on this distance metric during the neighborhood exploration and feeds *actual* SQL queries (and not just their column sets) into the existing designer. Internally, the existing designer compares the actual latency of different SQL queries, accounting for their different plans, joins, and all other details of every input query. For example, the designer ignores the less expensive queries to spend its budget on the more expensive ones.

Second, we must be able to efficiently sample the Γ-neighborhood of a given workload (see Algorithm 3, Line 2), which we can do when our cost function is $\delta_{euclidean}$. The sampling algorithm becomes computationally prohibitive when our distance metric involves computing the latency of

different queries. In Section 3.6, we thoroughly evaluate our `CliffGuard` algorithm overall, and our distance function in particular.

Inputs: W_0: an initial workload,
$\{\tilde{W}_1, \cdots, \tilde{W}_m\}$: workloads to merge with W_0,
f: the cost function (or its estimate),
D: a given design,
α: a scaling factor for the weight ($\alpha > 0$)
Output: W_{moved}: a new (merged) workload which is closer to $\{\tilde{W}_1, \cdots, \tilde{W}_N\}$ than W_0, i.e.,
$\Sigma_i \delta(\tilde{W}_i, W_{moved}) < \Sigma_i \delta(\tilde{W}_i, W_0)$

Subroutine `MoveWorkload` $(W_0, \{\tilde{W}_1, \cdots, \tilde{W}_m\}, f, D, \alpha)$
2 $W_{moved} \leftarrow \{\}$
3 $Q \leftarrow$ the set of all queries in W_0 and $\tilde{W}_1, \cdots, \tilde{W}_m$ workloads
4 **foreach** query $q \in Q$ **do**
5 $f_q \leftarrow f(\{q\}, D)$ **// the cost of query q using design D**
6 $\omega_q \leftarrow (f_q \cdot \sum_{i=1}^{m} \text{weight}(q, \tilde{W}_i))^\alpha + \text{weight}(q, W_0)$
7 $W_{moved} \leftarrow W_{moved} \cup \{(q, \omega_q)\}$
8 **return** W_{moved}

Algorithm 4: The subroutine for moving a workload.

The third, and final, reason is that the sole goal of our distance metric is to provide users a means to express and receive their desired degree of robustness. We show that despite its simplistic nature, $\delta_{euclidean}$ is still quite effective in satisfying (3.6) (see Section 3.6.3), and most importantly in enabling `CliffGuard` to achieve decisive superiority over existing designers (see Section 3.6.4).

3.6 Empirical Studies

In this section, we study `CliffGuard` empirically with an extensive set of experiments. The purpose of our experiments in this section is to demonstrate that (i) real world workloads can vary over time and be subject to a great deal of uncertainty (Section 3.6.2), (ii) despite its simplicity, our distance metric $\delta_{euclidean}$ can reasonably capture the performance implications of a changing workload (Section 3.6.3), and most importantly (iii) our robust design formulation and algorithm improve the performance of the state-of-the-art industrial designers by up to an order of magnitude, without having to modify the internal implementations of these commercial tools (Section 3.6.4). We also study different degrees of robustness (Section 3.6.5).

3.6.1 Experimental Setup

We have implemented `CliffGuard` in Java. We tested our algorithm against Vertica's database designer (called DBD [234]) and DBMS-X's designer as two of the most heavily-used state-of-the-art commercial designers, as well as two other baseline algorithms (introduced later in this section). For Vertica experiments, we used its community edition and invoked its DBD and query optimizer via a JDBC driver. Similarly, we used DBMS-X's latest API. We ran each experiment on two machines: a server and a client. The server ran a copy of the database and was used for testing different designs. The client was used for invoking the designer and sending queries to the server. We ran the Vertica experiments on two Dell machines running Red Hat Enterprise Linux 6.5, each with two quad-core Intel Xeon 2.10GHz processors. One of the machines had 128GB memory and 8×4TB 7.2K RPM disks (used as server) and the other had 64GB memory and 4×4TB 7.2K RPM disks. For DBMS-X experiments, we used two Azure Standard Tier A3 instances, each with a quad-core AMD Opteron 4171 HE 2.10GHz processor, 7GB memory, and 126GB virtual disks. In this section, when not specified, we refer to our Vertica experiments.

Workloads[8]— We conducted our experiments on a real-world (R1) workload and two synthetic ones (S1 and S2). R1 belongs to one of the largest customers of the Vertica database, composed of 310 tables and $430+K$ time-stamped queries issued between March 2011 and April 2012 out of which $15.5K$ queries conform to their latest schema (i.e., can be parsed). We did not have access to their original dataset but we did have access to their data distribution, which we used to generate a 151GB dataset for our Vertica experiments. Since we did not have access to any real workloads from DBMS-X's customers, we used the same query log but on a smaller dataset (20GB) given the smaller memory capacity of our Azure instances (compared to our Dell servers). We also created two synthetic workloads, called S1 and S2, as follows. We used the same schema and dataset as R1, but chose different subsets and relative ordering of R1 queries to artificially cause different degrees of workload change. Table 3.1 reports basic statistics on the amount workload changes (in terms of $\delta_{euclidean}$) between consecutive windows of queries where each window was 28 days (different window sizes are studied in Section 3.6.2). S1 queries were chosen to mimic a workload with minimal change over time (between $0.1m$ and m, where m is the minimum change observed in R1). S2 queries were chosen to exhibit the same range of $\delta_{euclidean}$ as R1 but more uniformly. More detailed analysis of these workloads will be presented in the subsequent sections.

Algorithms Compared— We divided the queries according to their timestamps into 4-week windows, $W_0, W_1, \cdots$. We re-designed the database at the end of each month to simulate a tuning frequency of a month (a common practice, based on our oral conversations). In other words, we

[8]Common benchmarks (e.g., TPC-H) are not applicable here as they only contain a few queries, and do not change over time.

Workload	**Min** $\delta(W_i,W_{i+1})$	**Max** $\delta(W_i,W_{i+1})$	**Avg** $\delta(W_i,W_{i+1})$	**Std** $\delta(W_i,W_{i+1})$
R1	m=0.00016	M=0.00311	0.00120	0.00122
S1	0.1m	m	0.00006	0.00003
S2	m	M	0.00178	0.00063

Table 3.1: Summary of our real-world and synthetic workloads.

fed W_i queries into each of the following designers and used the produced design to process W_{i+1} (except for FutureKnowingDesigner; see below).

1. NoDesign: A dummy designer that returns an empty design (i.e., no projections). Using NoDesign all queries simply scan the default super-projections (which contain all the columns), providing an upper limit on each query's latency.

2. ExistingDesigner: The nominal designer shipped with commercial databases. For instance, Vertica's DBD [234] recommends a set of projections while DBMS-X's designer finds various types of indices and materialized views. We used these state-of-the-art designers as our main baselines.

3. FutureKnowingDesigner: The same designer as ExistingDesigner, except that instead of feeding queries from W_i and testing on W_{i+1}, we both feed and test it on W_{i+1}. This designer signifies the best performance achievable where the designer knows exactly which queries to expect in the future and optimize for.

4. MajorityVoteDesigner: A designer that uses sensitivity analysis to identify elements of the nominal design that are *brittle* against changes of workload. This designer uses the same technique as CliffGuard to explore the local neighborhood of the current W_i, and generate a set of perturbed workloads $W_i^1, \cdots, W_i^n$. Then, it invokes the ExistingDesigner to suggest an optimal design for each W_i^j. Finally, for each structure (e.g., index, materialized view, projection) s, MajorityVoteDesigner counts the number of times that s has appeared in the nominal design of the neighbors, and selects those structures that have appeared in different designs most frequently. The idea behind this heuristic is that structures that appear in the optimal design of fewer neighbors (have fewer votes) are less likely to remain beneficial when the future workload changes.

5. OptimalLocalSearchDesigner: Similar to MajorityVoteDesigner, this designer starts by searching the local neighborhood of the given workload and generating perturbed workloads. However, instead of selecting structures that have been voted for by the most number of neighbors, this designer takes the union of the queries in the neighboring workloads as the *expectation* (i.e., representative) of the future workload, say $\bar{W}$. This algorithm then solves an Integer Linear Program to find an optimal set of structures that fit in the budget and minimize the cost of $\bar{W}$.[9]

[9] A greedy version of this algorithm and a detailed description of the other baselines can be found in our technical

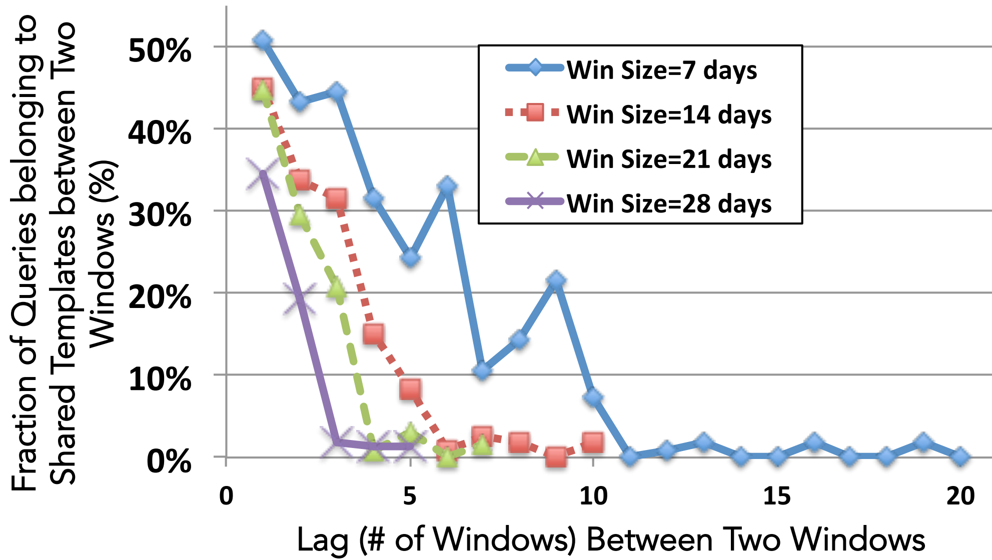


Figure 3.5: Many workloads drift over time ($15.5K$ queries, 6 months).

7. `CliffGuard`: Our robust database designer from Section 3.4.

Note that DBD and DBMS-X's designer (`ExistingDesigner`) are our goal standards as the state-of-the-art designers currently used in the industry. However, we also aim to answer the following question. How much of `CliffGuard`'s overall improvement over nominal designers is due to its exploration of the initial workload's local neighborhood, and how much is due to its carefully selected descent direction and step sizes in moving away from the worst neighbors? Since `MajorityVoteDesigner` and `OptimalLocalSearchDesigner` use the same neighborhood sampling strategy as `CliffGuard` but employ greedy and local search heuristics, we will be able to break down the contribution of `CliffGuard`'s various components to its overall performance.

Since Vertica automatically decides on the storage budget (50GB in our case), we used the same budget for the other algorithms too. For DBMS-X experiments, we used a maximum budget of 10GB (since the dataset was smaller). Also, unless otherwise specified, we used n=20 samples in all algorithms involving sampling, and 5 iterations, $\lambda_{success} = 5$, and $\lambda_{success} = 0.5$ in `CliffGuard`.

3.6.2 Workloads Change Over Time

First, we studied if and how much our real workload has changed over time. While OLTP and *reporting* queries tend to be more repetitive (often instantiated from a few templates with different parameters), analytical and exploratory workloads tend to be less predictable (e.g., Hive queries at Facebook are reported to access over 200–450 different subsets of columns [31]). Likewise, in

report [189].

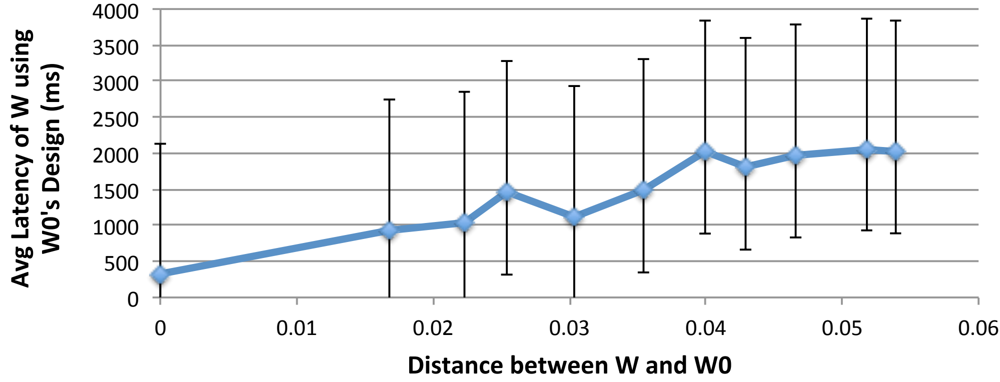


Figure 3.6: Performance decay of a window W on a design made for another window W_0 is highly correlated with their distance.

our analytical workload R1, we observed that queries issued by users have constantly drifted over time, perhaps due to the changing nature of their company's business needs.

Figure 3.5 shows the percentage of queries that belonged to templates that were shared among each pair of windows as the time lag grew between the two windows. Here, we have defined templates by stripping away the query details except for the sets of columns used in the `select`, `where`, `group by`, and `order by` clauses. This is an overly optimistic analysis assuming that queries with the same column sets in their respective clauses will exhibit a similar performance. However, even with this optimistic assumption, we observed that for a window size of one week, on average only 51% of the queries had a similar counterpart between consecutive weeks. This percentage was only 35% when our window was 4 weeks. Regardless of the window size, this commonality drops quickly as the time lag increases, e.g., after 2.5 months less than 10% of the queries had similar templates appearing in the past. The unpredictability of analytical workloads underlines the important role of a robust designer. We show in Section 3.6.4 that failing to take into account this potential change (i.e., uncertainty) in our target workload has a severe impact on the performance of existing physical designers — one that we aim to overcome via our robust designs.

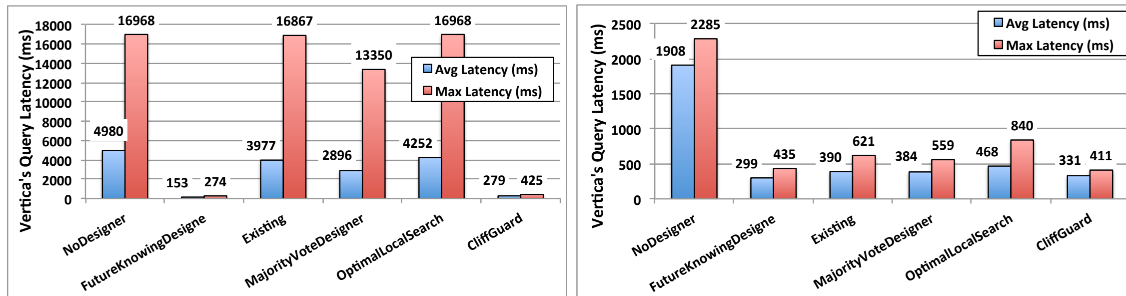
3.6.3 Our Distance Metric Is Sound

In Section 3.5, we introduced our distance metric $\delta_{euclidean}$ to concisely quantify the dissimilarity of two SQL workloads. While we do not claim that $\delta_{euclidean}$ is an ideal one (see Section 3.5), here we show that it is sound. That is, in general:

$$\delta(W_0, W) \leq \delta(W_0, W') \Rightarrow f(W, \mathbb{D}(W_0)) \leq f(W', \mathbb{D}(W_0))$$

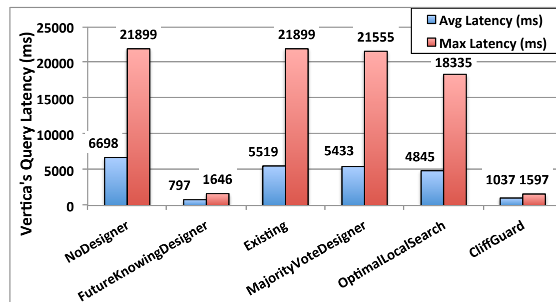
which means that a design made for W_0 is more suitable for W than it is for W', i.e., W will experience a lower latency than W'. Figure 3.6 reports an experiment where we chose 10 different

starting windows as our W_0 and created a number of windows with different distances from W_0. The curve (error bar) shows the average (range) of the latencies of these different windows for each distance. This plot indicates a strong correlation and monotonic relationship between performance decay and $\delta_{euclidean}$. Later, in Section 3.6.4, we show that even with this simplistic distance metric, our `CliffGuard` algorithm can consistently improve on Vertica's latest designer by severalfold.



(a) Real-world workload R1 on Vertica.

(b) Synthetic static workload S1 on Vertica.



(c) Synthetic static workload S2 on Vertica.

Figure 3.7: Average and worst-case performances of designers for Vertica, averaged over all windows, for workloads R1, S1, and S2.

3.6.4 Quality of Robust vs. Nominal Designs

In this section, we turn to the most important questions of this chapter: is our robust designer superior to state-of-the-art designers? And, if so, by what measure? We compared these designers using all 3 workloads. In R1, out of the 15.5K queries, only 515 could benefit from a physical design, i.e., the remaining queries were either trivial (e.g., `select version()`) or returned an entire table (e.g., '`select * from T`' queries with no filtering used for backup purposes) in which case they always took the same time as they only used the super-projections in Vertica and table-scans in DBMS-X. Thus, we only considered queries for which there existed an *ideal* design (no matter how expensive) that could improve on their bare table-scan latency by at least a factor of $3\times$.

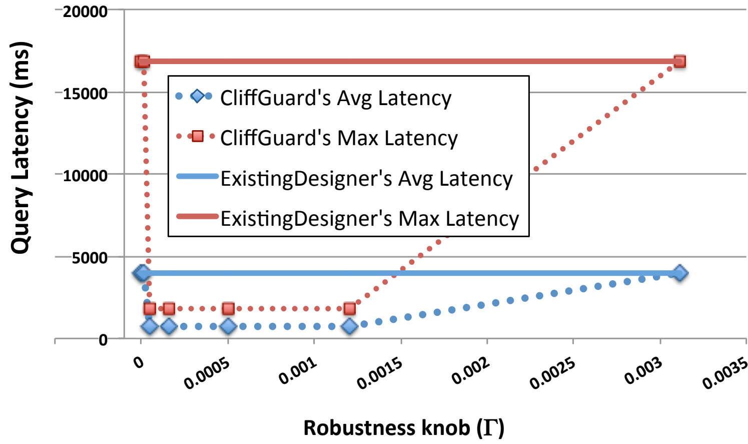


Figure 3.8: Different degrees of robustness for Workload R1.

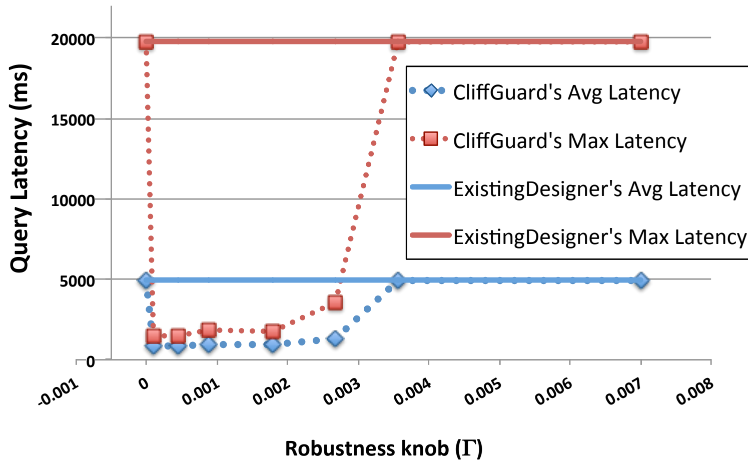


Figure 3.9: Different degrees of robustness for Workload S2.

Figure 3.7 summarizes the results of our performance comparison on Vertica, showing the average and maximum latencies (both averaged over all windows) for all three workloads. On average, `MajorityVoteDesigner` improved on the existing designer by 13%, while `OptimalLocalSearchDesigner`'s performance was slightly worse than Vertica's DBD. However, `CliffGuard` was superior to the existing designer by an astonishing margin: on average, it cut down the maximum latency of each window by $39.7\times$ and $13.7\times$ for R1 and S2, respectively. Interestingly, for these workloads, even `CliffGuard`'s average-case performance was $14.3\times$ and $5.3\times$ faster than `ExistingDesigner`. The last result is surprising because our `CliffGuard` is designed to protect against worst-case scenarios and ensure a predictable performance. However, improvement even on the average case indicates that the design space of a database is highly non-convex — and as such can easily delude a designer into a local optimum. Thus, by avoiding the proximity of bad neighbors, `CliffGuard` seems to find designs that are also more globally optimal. In fact, for S2,

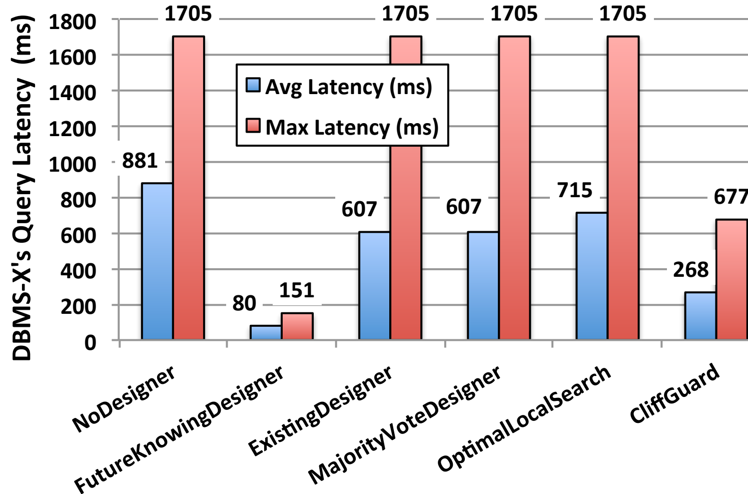


Figure 3.10: Performance of different designers for DBMS-X on workload R1.

Figure 3.7c shows that `CliffGuard` is only 30% worse than a hypothetical, ideal world where future queries are precisely known in advance (i.e., the `FutureKnowingDesigner`). For S1, however, `CliffGuard`'s improvement over `ExistingDesigner` is more modest: $1.5\times$ improvement for worst-case latency and $1.2\times$ for average latency. This is completely expected since S1 is designed to exhibit no or little change between different windows (refer to Table 3.1). This is the ideal case for a nominal designer since the amount of uncertainty across workloads is so negligible that even our hypothetical `FutureKnowingDesigner` cannot improve much on the nominal designer. Thus, averaging over all three workloads, compared to `ExistingDesigner`, `CliffGuard` improves the average and worst-case latencies by $6.9\times$ and $18.3\times$, respectively.

Figure 3.10 reports a similar experiment for workload R1 but for DBMS-X. Even though DBMS-X's designer has been fine-tuned and optimized over the years, `CliffGuard` still improves its worst-case and average-case performances by 2.5–$5.2\times$ and 2–$3.2\times$, respectively. This is quite encouraging given that `CliffGuard` is still in its infancy stage of development and treats the database as a black-box. While still significant, the improvements here are smaller than those observed with Vertica. This is due to several heuristics used in DBMS-X's designer (such as omitting workload details) that prevent it from overfitting its input workload. However, this also shows that dealing with such uncertainties in a principled framework can be much more effective.

These experiments confirm our hypothesis that failing to account for workload uncertainty can have significant consequences. For example, for R1 on Vertica, `ExistingDesigner` is on average only 25% better than `NoDesign` (with no advantage for the worst-case). Note that here the database was re-designed every month, which means even this slight advantage of `ExistingDesigner` over `NoDesign` would quickly fade away if the database were to be re-designed less frequently (as the distance between windows often increases with time; see Figure 3.5). These experiments show the

ample importance of re-thinking and re-architecting the existing designers currently shipped and used in our database systems.

3.6.5 Effect of Robustness Knob on Performance

To study the effect of different levels of robustness, we varied the Γ parameter in our algorithm and measured the average and worst-case performances in each case. The results of this experiment for workloads R1 and S2 are shown in Figures 3.8 and 3.9, respectively. (As reported in Section 3.6.4, workload S1 contains minimal uncertainty and thus is ruled out from this experiment, i.e., the performance difference between `ExistingDesigner` and `CliffGuard` remains small for S1). Here, experiments on both workloads confirm that requesting a large level of robustness will force `CliffGuard` to be overly conservative, eliminating its margin of improvement over `ExistingDesigner`. Note that in either case `CliffGuard` still performs no worse than `ExistingDesigner`, which is due to two reasons. First, `ExistingDesigner` is only marginally better than `NoDesign` (refer to Section 3.6.4) and as Γ increases, its relevance for the actual workload (which has a much lower $\delta_{euclidean}$) degrades. As a result, both designers approach `NoDesign`'s performance, which serves an upper bound on latency (i.e., unlike theory, latencies are always bounded in practice, due to the finite cost of the worst query plan). The second reason is that, during each iteration of `CliffGuard` (unlike BNT), our new workload always contains the original workload which ensures that even when Γ is large, the designer will not completely ignore the original workload (see Algorithm 4). Also, as expected, as Γ approaches zero, `CliffGuard`'s performance again approaches that of a nominal designer.

3.7 Related Work

There has been much research on physical database design problems, such as the automatic selection of materialized views [140, 174, 217, 250], indices [74, 179, 197, 231], or both [36, 91, 159, 259]. Also, most modern databases come with designer tools, e.g., Tuning Wizard in Microsoft SQL Server [36], IBM DB2's Design Advisor [259], Teradata's Index Wizard [59], and Oracle's SQL Tuning Adviser [91]. Other types of design problems include project selection in columnar databases [100, 163, 234], stratified sample selection in approximate databases [26, 31, 42, 66], and optimizing different replicas for different workloads in a replicated databases [229]. All these designers are nominal and assume that their target workload is precisely known. Since future queries are often not known in advance, these tools optimize for past queries as approximations of future ones. By failing to take into account the fact that a portion of those queries will be different in the future, they produce designs that are sub-optimal and brittle in practice. To mitigate

some of these problems, a few heuristics [76] have been proposed to compress and summarize the workload [69, 158] or modify the query optimizer to produce richer statistics [113]. However, these approaches are not principled and thus, do not necessarily guarantee robustness. In contract, CliffGuard takes the possible changes of workload into account in a principled manner, and directly maximizes the robustness of the physical design.

To avoid these limitations, adaptive indexing schemes [119, 120, 128, 143, 215]) take the other extreme by avoiding the offline physical design, and instead, creating and adjusting indices incrementally, *on demand*. Despite their many merits, these schemes do not have a mechanism to incorporate prior knowledge under a bounded amount of uncertainty. Also, one still needs to decide which subsets of columns to build an adaptive index on. For these reasons, most commercial databases still rely on their offline designers. In contrast, CliffGuard uses RO theory to directly minimize the effect of uncertainty on optimality, and guarantee robustness.

The effect of uncertainty (caused by cost and cardinality estimates) has also been studied in the context of query optimization [41, 70, 82, 103, 181, 206] and choosing query plans with a bounded worst-case [39]. None of these studies have addressed uncertainties caused by workload changes, or their impact on physical designs. Also, while these approaches produce plans that are more predictable, they are not principled in that they do not directly maximize robustness, i.e., they do not guarantee robustness even in the context of query optimization. Finally, most of these heuristics are specific to a particular problem and do not generalize to others.

Theory of robust optimization has taken many strides in recent years [51, 52, 53, 78, 102, 166, 255] and has been applied to many other disciplines, e.g., supply chain management [51], circuit [199] and antenna [175] design, power control [136], control theory [50], thin-film manufacturing [55], and microchip architecture [195].

3.8 Summary

In this chapter, we empirically validated the effectiveness of a robust designer, CliffGuard. Robust designs enable databases to be more resilient against changes of the workload and their noisy environments. Thus, they do not need to be re-tuned or re-designed at the same frequency as other databases that use nominal design. This can dramatically reduce the operational cost of database administration, which both frees up critical personnel to work on novel computational tasks, and saves the resources of vendors and organizations alike. In addition, since robust designs ensure that databases deliver a highly consistent performance, they also enable new mission-critical applications that require a predictable and reliable service, across commerce, science, and government.

CHAPTER 4

Joins on Samples: A Hybrid Sampling Scheme with Optimal Sampling Strategy

4.1 Motivation

Approximate query processing (AQP) has regained significant attention in recent years due to major trends in the industry. Larger datasets and the rise of shared and hosted infrastructure have made it more expensive to achieve interactive-speed analytics. AQP presents itself as a viable alternative in scenarios where perfect decisions can be made with imperfect answers [31]. AQP is most appealing when negligible loss of accuracy can be traded for a significant gain in speedup or computational resources. Adhoc analytics [223], visualization [90, 156, 204, 219, 258], IoT [10], A/B testing [30], email marketing and customer segmentation [118], and real-time threat detection [9] are examples of such usecases.

Sampling and Joins— Sampling is one of the most widely-used techniques for general-purpose AQP [87]. The high level idea is to execute the query on a small sample of the original table(s) in order to provide a fast, but approximate, answer. While effective for simple aggregates, using samples for join queries has long remained an open problem [28]. There are two main approaches to AQP: *offline* or *online*. Offline approaches [27, 29, 32, 67, 110, 198] build samples (or other synopses) prior to query arrival. At run time, they simply choose appropriate samples that can yield the best accuracy/performance for each incoming query. Online approaches, on the other hand, perform much of their sampling at run time based on the query at hand [43, 85, 133, 152, 196, 243]. Naturally, offline sampling leads to significantly higher speedup, while online techniques can support a much wider class of queries [152]. The same taxonomy applies to join approximation: offline techniques perform joins on previously-prepared samples [29, 71, 79, 198, 256], while online approaches seek to produce a sample of the output of the join at run time [125, 146, 169, 176]. As mentioned, the latter often means more modest speedups (e.g., 2× [152]) which may not be sufficient to justify approximation, or additional requirements (e.g., an index for each join col-

umn [169]) which may not be acceptable to many applications. Thus, our focus in this chapter—and what is considered an open-problem—is the offline approach: joins on samples, not sampling the join's output.

Joins on Samples— The simplest strategy is as follows. Given two large tables T_1 and T_2, create a uniform random sample of each, say S_1 and S_2 respectively, and then use $S_1 \bowtie S_2$ to approximate aggregate statistics of $T_1 \bowtie T_2$. This will lead to significant speedup if samples are much smaller than original tables, i.e., $|T_i| >> |S_i|$.

One of the earliest results in this area shows that this simple strategy is futile for two reasons [115]. First, joining two uniform samples leads to quadratically fewer output tuples, i.e., joining two uniform samples that are each p fraction ($0 \leq p < 1$) of the original tables will only produce p^2 of the output tuples of the original join (see Figure 4.1). Second, joining uniform samples of two tables does not yield an independent sample of the join of those tables (see Section 4.2.1 for details).[1] The dependence of the output tuples can drastically lower the approximation accuracy [71, 115].

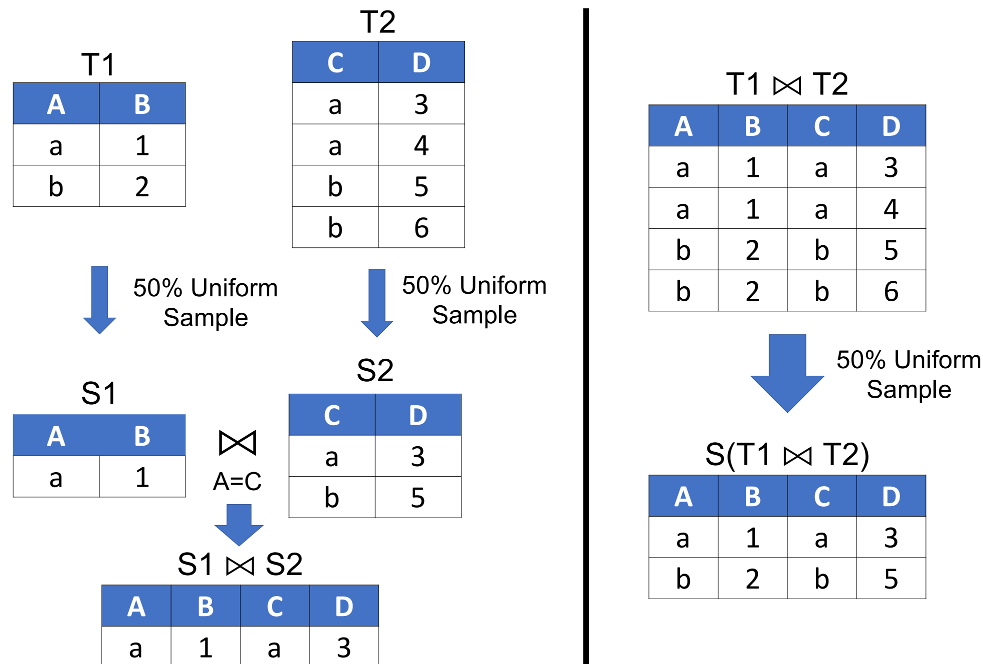


Figure 4.1: A toy example of joining two uniform samples (left) versus a uniform sample of the join (right).

[1]Prior work has stated this, as joining uniform samples is not a *uniform* sample of the join [29]. We avoid this terminology because uniform means equal probability of inclusion, and in this case each tuple does appear in the join of the uniform samples with equal probability, but not independently. In other words, joining two i.i.d. samples is an identical, but not independent, sample of the join tuples.

Prior Work— *Universe* sampling [127, 152, 198] addresses the first drawback of uniform sampling. Although universe sampling avoids quadratic reduction of output, its creates even more correlation in its output, leading to much lower accuracy (see Section 4.3.1).

Atserias et al. show that computing exact joins with a small memory or time budget is hard [40], by providing a worst case lower bound for any query involving equi-joins on multiple relations. For instance, the maximum possible join size for any cyclic join on three n-tuple relations is $\Theta(n^{1.5})$. Thus, a natural question is whether approximating joins is also hard with small memory or time.

Our Goal— In this chapter, we specifically focus on understanding the limitation of using offline samples in approximating join queries. Given a sampling budget, how well can we approximate the join of two tables using their offline samples? To answer this question, we must first define what constitutes a "good" approximation of a join. We consider two metrics: (1) output cardinality and (2) aggregation accuracy. The former is the number of tuples of the original join that also appear in the join of the samples, whereas the latter is the error of the aggregates estimated from the sample-based join with respect to their true values, if computed on the original join. Because in this chapter we only consider unbiased estimators, we measure approximation error in terms of the variance of our estimators.

For the first metric, we provide a simple proof showing that universe sampling is optimal from [137], i.e. no sampling scheme with the same sampling rate can outperform universe sampling in terms of the (expected) output cardinality. However, as we describe in Section 4.3.1, retaining a large number of join tuples does not imply accurate aggregates. It is therefore natural to also ask about the lowest variance that can be achieved given a sampling rate. We summarize an information-theoretical lower bound to this question, derived in [137]. We also introduce a hybrid sampling scheme that matches this lower bound within a constant factor. This scheme involves a centralized computation, which can become prohibitive for large tables due to large amounts of statistics that need to be shuffled across the network. Thus, a decentralized variant is also proposed. This only shuffles a minimal amount of information across the nodes—such as the table size and maximum frequency—but still achieves the same worst case guarantees. Finally, we generalize our sampling scheme to accommodate *a priori* information about filters (i.e., `WHERE` clause).

In this chapter, we make the following contributions:

1. We discuss two metrics—output size and estimator's variance —for measuring the quality of join approximation, and show that universe sampling is optimal for output size and there is an information-theoretical lower bound for variance (Section 4.3).

2. We summarize a hybrid sampling scheme [137], called Stratified-Universe-Bernoulli Sampling (SUBS), which allows for different combinations of stratified, universe, and Bernoulli sampling. We describe optimal sampling parameters within this scheme, which achieve the theoretical

lower bound of variance within a constant factor (Section 4.4–4.5.3). We also describe its extension to accommodate additional information regarding the `WHERE` clause (Section 4.6).

3. Through extensive experiments, we empirically show that SUBS with the optimal sampling parameters achieve lower error than existing sampling schemes in both centralized and decentralized scenarios (Section 4.7).

4.2 Background

In this section, we provide the necessary background on sampling-based join approximation. We describe the problem setting and assumptions in [137].

4.2.1 Sampling in Databases

The following are the three main popular sampling strategies (operators) used in AQP engines and database systems.

1. **Uniform/Bernoulli Sampling.** Any strategy that samples all tuples with the same probability is considered a uniform (random) sample. Since enforcing fixed-size sampling without replacement is expensive in distributed systems, Bernoulli sampling is considered a more efficient strategy [152]. In Bernoulli sampling, each tuple is included in the sample independently, with a fixed sampling probability p. In this chapter, for simplicity, we use "uniform" and "Bernoulli" interchangeably. As mentioned in Section 4.1, joining two uniform samples leads to quadratically fewer output tuples. Further, it does not guarantee an i.i.d. sample of the original join [29]: the output is a uniform sample of the join but not an independent one. Consider an arbitrary tuple of the join, say (t_1, t_2) where t_1 is from the first table and t_2 is from the second one. The probability of this tuple appearing in the join of the samples is always the same value, i.e., p^2. The output is therefore a uniform sample. However, the tuples are not independent: consider another tuple of the join, say (t_1, t_2') where t_2' is another tuple from the second table joining with t_1. If (t_1, t_2) appears in the output, the probability of (t_1, t_2') also appearing becomes p instead of p^2, which would be the probability if they were independent.

2. **Universe Sampling.** Given a column[2] J, a (perfect) hash function $h : J \mapsto [0, 1]$, and a sampling rate p, this strategy includes a tuple t in the table if $h(t.J) \leq p$. This strategy is often used for equi-joins, in which case the same p value and hash function h are applied to the join columns in both tables. This ensures that when a tuple t_1 is sampled from one table, any matching tuple t_2 from the other table is also sampled, simply because $t_1.J = t_2.J \Leftrightarrow h(t_1.J) = h(t_2.J)$. This is

[2] J can also be a set of multiple columns.

why joining two universe samples with a sampling rate of p produces p fraction of the original join output *in expectation*. The output is a uniform sample of the original join, as each join tuple appears with the same probability p. However, there is more dependence among the output tuples. Consider two join tuples (t_1, t_2) and (t'_1, t'_2) where t_1, t'_1, t_2, t'_2 all share the same join key. Then, if (t_1, t_2) appears, the probability of (t'_1, t'_2) also appearing will be 1. Likewise, if (t_1, t_2) does not appear, the probability of (t'_1, t'_2) appearing will be 0. Higher dependence means lower accuracy (see Section 4.3.1).

3. **Stratified Sampling.** There are different variants of stratified sampling. The key idea is to ensure sample representation for each distinct value of a set of columns C, called stratified columns. Given the stratified columns C and a target frequency k_{key}, one simple strategy is to choose t tuples uniformly at random from each group of tuples with the same value of C [31]. When a group has fewer than k_{key} tuples, all of them are retained.

4.2.2 Quality Metrics

Different metrics can be used to assess the quality of a join approximation. In this chapter, we focus on the following two, which are used by most AQP systems.

Output Size/Cardinality— This metric is the number of tuples of the original join that also appear in the join of the samples. This metric is mostly relevant for exploratory usecases, where users visualize or examine a subset of the output. In other cases, where an aggregate is computed from the join output, retaining a large number of output tuples does not guarantee accurate answers (we show this in Section 4.3.1).

Variance— In scenarios where an aggregate function needs to be calculated from the join output, the error of the aggregate approximation is more relevant than the number of intermediate tuples generated. For most non-extreme statistics, there are readily available unbiased estimators, e.g., Horvitz-Thompson estimator [135]. Thus, a popular indicator of accuracy is the variance of the estimator [31], which determines the size of the confidence interval given a sample size.

4.2.3 Problem Statement

In this section, we formally state the problem of sample-based join approximation. The notations used throughout the chapter are listed in Table 4.1.

Query Estimator— Let S_1 and S_2 be two samples generated offline from tables T_1 and T_2, respectively, and q_{agg} be a query that computes an aggregate function agg on the join of T_1 and T_2. A query estimator $\hat{J}_{agg}(S_1, S_2)$ is a function that estimates the value of agg using two samples rather

Notation	Definition
T_1, T_2	Two tables for the join
S_i	A sample generated from table T_i
J	Column(s) used for the join between T_1 and T_2
W	Column being aggregated (e.g., `SUM`, `AVG`)
C	Column(s) used for filters (i.e., `WHERE` clause)
$\mathcal{U}$	Set of all possible values of J
a, b	Frequency vectors for T_1 and T_2 w.r.t. its join column resp.
a_v, b_v	Number of tuples with join value v in T_1 and T_2, resp.
$\hat{J}_{agg}$	Estimator for a join query with aggregate function agg
ϵ	Sampling budget w.r.t. the original table size
n_1, n_2	Number of tuples in T_1 and T_2, resp.
h	A (perfect) hash function
k_{tuple}	minimum number of tuples to be kept per group in stratified sampling
k_{key}	minimum number of join keys per group to apply universe sampling
p	Sampling rate of universe sampling
q	Sampling rate of uniform sampling

Table 4.1: Notations.

than the original tables.

Join Sampling Problem— Given a query estimator $\hat{J}_{agg}$ and a sampling budget $\epsilon \in (0, 1]$, our goal is to create an optimal pair of samples S_1 and S_2—from tables T_1 and T_2, respectively— that are optimal in terms of a given success metric, while respecting a given storage budget $epsilon$ on average. Specifically, we seek S_1 and S_2 that minimize $\hat{J}_{agg}$'s variance or maximize its output size such that $E[|S_1| + |S_2|] \leq \epsilon \times (|T_1| + |T_2|)$.

Note that we define the sampling budget in terms of an expected size (rather than a strict one), since most sampling schemes are probabilistic in nature and may slightly over- or under-use a given budget.

To formally study this problem, we first need to define a class of reasonable sampling strategies. In Section 4.4, we define a hybrid scheme that can capture different combinations of stratified, universe, and uniform sampling.

4.2.4 Scope and Limitations

To simplify our analysis, we limit our scope in this chapter.

Flat Equi-joins— We focus on equi (inner) joins as the most common form of joins in practice. We also support both WHERE and GROUPBY clauses. Because our focus is on the join itself, we ignore nested queries and only consider flat (or flattened) queries. We primarily focus on two-way joins. However, our results extend to multi-way joins with the same join column(s).

Aggregate Functions— Most AQP systems do not support extreme statistics, such as Min or Max [190]. Likewise, we only consider non-extreme aggregates, and primarily focus on the three basic functions, COUNT, SUM, and AVG. However, we expect our techniques to easily extend to other mean-like statistics as well, such as VAR, STDEV, and PERCENTILE.

4.3 Hardness

In this section, we explain why providing a large output size is insufficient for approximating joins, and show the hardness of approximating common aggregates based on the theory of communication complexity from [137].

4.3.1 Output Size

Uniform sampling leads to small output size. If we sample at a rate q from both table T_1 and table T_2, the join of samples contains only q^2 fraction of $T_1 \bowtie T_2$ in expectation. Moreover, the join of two independent samples of the original tables is in general not an independent sample of $T_1 \bowtie T_2$, which hurts the sample quality. In contrast, universe sampling [127, 152] with sample rate p can in expectation sample a p fraction of $T_1 \bowtie T_2$. We prove that this is optimal (all omitted proofs are deferred to [137]).

Theorem 1. *Any sampling scheme with sample rate α can sample, at most, α fraction of $T_1 \bowtie T_2$ in expectation in the worst case input.*

However, a large number of tuples retained in the join does not imply that the original join query can be accurately approximated. As pointed out in [79], universe sampling shows poor performance in approximating queries when the frequencies of keys are concentrated on a few elements. Consider the following extreme example with tables T_1 and T_2, each comprised of n tuples with a single value 1 in their join key. In this example, universe sampling with the sampling rate p produces an estimator of variance n^4/p, while uniform sampling with rate q has a variance of n^2/q^2, which is much lower when $p = q$ and n is large. Thus, a larger output size does not necessarily lead to a better approximation of the query.

4.3.2 Approximating Aggregate Queries

In this section, we focus on the core question: why is approximating common aggregates (e.g., COUNT, SUM and AVG) hard when using a small sample (or more generally, a small summary)? We address this question using the theory of communication complexity. Specifically, to show that computing COUNT on a join is hard, we reduce it to set intersection, a canonically hard problem in communication complexity. Assume that both Alice and Bob each hold a set of size k, say A and B, respectively. They aim to estimate the size of $t = |A \cap B|$. Pagh et. al [194] show that if Alice only sends a small summary to Bob, any unbiased estimator that Bob uses will have a large variance.

Theorem 2 (See [194])**.** *Any one-way communication protocol that estimates t within relative error δ with probability at least $2/3$ must send at least $\Omega(k/(t\delta^2))n$ bits.*

Corollary 3. *Any estimator to $|A \cap B|$ produced by Bob that is based on an s-bits summary by Alice must have a variance of at least $\Omega(kt/s)$.*

Any sample of size s can be encoded using $O(\log \binom{k}{s})$ bits, implying that any estimator to COUNT that is based on a sample of size s from one of the tables must have a variance of at least $\Omega(kt/s)$.

Estimating SUM queries is at least as hard as estimating COUNT queries, since any COUNT can be reduced to a SUM by setting all entries in the SUM column to 1.

From the hard instance of set intersection, we can also derive a hard instance for AVG queries. Based on Theorem 2, any summary of T_1 that can distinguish between intersection size $t(1+\delta)$ and $t(1-\delta)$ must be at least of size $\Omega(k/(t\delta^2))$ bits. Now we reduce this problem to estimating an AVG query.

Here, the two tables consist of $k + \sqrt{t}$ tuples each. The first k tuples of T_1 and T_2 are from the hard instance of set intersection, and the values of their AVG column are set to $2r$. The join column of the last $\sqrt{t}$ tuples is set to some common key v' that is in the first k tuples, and their AVG column is set to 0. Therefore, the intersection size from the first k tuples is at least $t(1+\delta)$ (or at most $(t(1-\delta))$) if and only if the result of the AVG query is at least $\frac{2rt(1+\delta)}{t(2+\delta)} = (1 + O(\delta))r$ (or at most $\frac{2rt(1+\delta)}{t(2+\delta)} = (1 - O(\delta)r)$). By re-scaling δ by a constant factor, we can get the following theorem:

Theorem 4. *Any summary of T_1 that can estimate an AVG query with precision δ with probability at least $2/3$ must have a size of at least $\Omega(n/(t\delta^2))$.*

4.4 Generic Sampling Scheme

To formally argue about the optimality of a sampling strategy, a *class* of sampling schemes must be defined first. As discussed in Section 4.2.1, there are three well-known sampling operators: stratified, universe, and Bernoulli (uniform). However, these atomic operators can themselves be combined. For example, one can apply universe sampling of rate 0.1 and then Bernoulli sampling of rate 0.2 for an overall effective sampling rate of 0.02.[3] To account for such hybrid schemes, [137] define a generic scheme that combines universe and Bernoulli sampling, called UBS. A more generic scheme is also defined that combines all three of stratified, universe and Bernoulli sampling, called SUBS. It is easy to show that the basic sampling operators are a special case of SUBS. First, we define the effective sample rate.

Definition 5 (Effective sampling rate)**.** *We define the effective sampling rate of a sampling scheme as the expected ratio of the size of the resulting sample to that of the original table.*

Definition 6 (Universe-Bernoulli Sampling (UBS) Scheme)**.** *Given a table T and a column (or set of columns) J in T, a UBS scheme is defined by a pair (p, q), where $0 < p \leq 1$ is a universe sampling rate and $0 < q \leq 1$ is a Bernoulli (or uniform) sampling rate. Let $h : \mathcal{U} \mapsto [0, 1]$ be a perfect hash function. Then, a sample of T produced by this scheme, $S = UBS_{p,q}(T, J)$, is produced as follows:*

function UBS$_{p,q}$(T, J)
  Initialize $S \leftarrow \emptyset$ **for** *all tuples t in T* **do**
    if $h(t.J) < p$ **then**
      Include t in S independently with probability q
  return S

It is easy to see that the effective sampling rate of a UBS scheme (p, q) is $p \cdot q$. Thus, the effective sampling rate of is independent of the actual distribution of the values in the table (and column(s) J).

The goal of this sampling paradigm is to optimize the trade-off between universe sampling and Bernoulli sampling in different instances. At one extreme, when each join value appears exactly once in both table, universe sampling leads to lower variance than Bernoulli sampling. This is because independent Bernoulli sampling has trouble matching tuples with the same join value, while universe sampling guarantees that when a tuple is sampled, all matching tuples in other table are also sampled. At the other extreme, if all tuples have the same join value in both tables (i.e.,

[3] Note that applying Bernoulli sampling before universe sampling defeats the purpose of the latter. This is because universe sampling aims to keep as many tuples with the same join key as possible, while with Bernoulli sampling will eliminate the majority of such tuples.

the join becomes a Cartesian product of the two tables), universe sampling will either sample the entire join, or sample nothing at all, while uniform sampling will have a sample size concentrated around qN, thus giving an estimator of much lower variance. In section 4.5.1 to 4.5.3, we give a comprehensive discussion on how to optimize p and q for different tables and different queries.

The Stratified-Universe-Bernoulli Sampling Scheme applies to a table T that is divided into K groups (i.e., strata), denoted as $G_1, G_2, \ldots, G_k$.

Definition 7 (Stratified-Universe-Bernoulli Sampling (SUBS) Scheme)**.** *Given a table T of N rows and a column (or set of columns) J in T, a SUBS scheme is defined by a tuple $(p_1, p_2, \ldots, p_K, q_1, q_2, \ldots, q_K)$, where $0 < p_i, q_i \leq 1$ are the universe sampling rate and Bernoulli sampling rate. Let $h : \mathcal{U} \mapsto [0, 1]$ be a perfect hash function. Then, a sample of T produced by this scheme, $S = UBS_{p,q}(T, J)$, is produced as follows:*

```
function SUBS_{p,q_1,...,q_K}(T, J, G)
    Initialize S ← ∅ for each group G_i do
        for all tuples t in G_i do
            if h(t.J) < p_i then
                Include t in S independently w/ prob. q_i
    return S
```

Let $|G_i|$ denote the number of tuples in group G_i. Then the effective sampling rate of a SUBS scheme is $\sum_i p_i \cdot q_i \cdot |G_i|/N$. We call $\epsilon_i = p \cdot q_i$ the effective sampling rate for group G_i.

In both UBS and SUBS schemes, the user specifies ϵ as his/her desired sampling budget, given which our goal is to determine optimal sampling parameters p and q (or p_i and q_i values) such that the variance of our join estimator is minimized. In Section 4.5, we derive the optimal p and q for UBS. For SUBS, in addition to ϵ, the user also provides two additional parameters k_{key} and k_{tuple} (explained below). Next, we show how to determine the effective sampling rate ϵ_i for each group G_i based on these parameters in SUBS. Given ϵ_i for each group, the problem is then reduced to finding the optimal parameters for UBS for that group (i.e., p_i and q_i). Moreover, as we will show in Sections 4.5.1–4.5.3, particularly in Lemma 10, the universe sampling rate for every group must be the same, and must be the same as the universe sampling rate of the other table in two-way joins. Hence, we use a single universe sampling rate $p = p_1 = \ldots = p_k$ across all groups.

As mentioned in Section 4.2.1, k_{tuple} is a user-specified lower bound on the minimum number of tuples[4] in each group the sample must retain. k_{key} is an additional user-specified parameter required for the SUBS scheme. It specifies a threshold as when to activate the universe sampler. In particular, if a group contains too few (i.e., less than k_{key}) *join keys*, we do not perform any universe sampling as it will have a high chance of filtering out all tuples. Hence, we apply universe

[4]The lower bound holds only on average, due to the probabilistic nature of sampling.

sampling only to those groups with $\geq k_{\text{key}}$ join keys. For groups with fewer than k_{key} join keys, we will only apply Bernoulli sampling with rate ϵ_i.

We call a group *large* if it contains at least s join keys, otherwise, we call it a *small* group. We use N_b to denote the total number of tuples in all large groups, and N_s to denote the total number of tuples in all small groups. Similarly, let M_b and M_s denote the number of large and small groups, respectively. Then, we decide the sampling budget ϵ_i for each group G_i as follows:

1. If $M_b t > \epsilon N_s$ or $M_b t > \epsilon N_b$, we notify the user that creating a sample given their parameters is infeasible.
2. Otherwise,
   - Let $\epsilon'_s = K_s \cdot t/N_s$ and let $\epsilon''_s = \epsilon - \epsilon'_s$. Then for each smallgroup G_i, the sampling budget is $\epsilon_i = t/|G_i| + \epsilon''_s$.
   - Let $\epsilon'_b = K_b \cdot t/N_b$ and let $\epsilon''_b = \epsilon - \epsilon'_b$. Then for each large group G_i, the sampling budget is $\epsilon_i = t/|G_i| + \epsilon''_b$.

One ϵ_i is determined for each group, the problem of deciding optimal SUBS parameters is reduced to deciding the optimal SUBS parameters for K separate groups. This effective sampling rate ϵ_i guarantees that each large group will have at least t tuples in the sample on average, and the remaining budget is divided evenly. Thus, the corresponding uniform sampling rate for each large group is $q_i = \epsilon_i/p$. Moreover, we pose the constraint that the universe sampling rate p should be at least $1/s$ to guarantee that, on average, there is at least one join key passing through the universe sampler.

For small groups, we simply perform a uniform sampling of rate ϵ_i. Conceptually, this is equivalent to setting $p = 1$ for these groups.

Overall, this strategy provides the following guarantees:

1. Each group will have at least t tuples in the sample, on average.
2. The probability of each group being missed is at most $(1 - 1/s)^s < 0.367$. In general, if we set $p>c/s$ for some constant $c>1$, this probability will become 0.367^c.
3. The approximation of the original query will be optimal in terms of its variance (see Sections 4.5.1–4.5.3).

4.5 Optimal Sampling

In this section, we summarize the derivation of optimal parameters for SUBS in [137]. As shown in Section 4.4, finding the optimal sampling parameters within the SUBS scheme can be reduced to that within the UBS scheme. Thus, in this section, we focus on providing a summary of deriving the UBS parameters that minimize error for each aggregation type (`COUNT`, `SUM`, and `AVG`). Initially,

we also assume there is no `WHERE` clause. Later, in Section 4.6, we show how to handle `WHERE` conditions and how to create a single sample instead of creating one per each aggregation type and `WHERE` condition.

Centralized vs. Decentralized— For each aggregation type, we analyze two scenarios: centralized and decentralized. Centralized setting is when the frequencies of the join keys in both tables are known. This represents situations where both tables are stored on the same server, or each server communicates its frequency statistics to other parties. Decentralized setting represents a scenario where the two tables are each stored on a separate server, and transmitting full frequency statistics across the network is costly.[5] In other words, each server only has access to full statistics of its own table (e.g., frequencies, join column distribution). However, as a minimal information, we assume the servers know each other's sampling budgets (ϵ_1 and ϵ_2).

4.5.1 Join Size Estimation: Count on Joins

We start by considering the following simplified query:

```
select count(*)
from T1 join T2 on J
```

where T_1 and T_2 are two tables joined on column(s) J. Consider two samples, $S_1 = \text{UBS}_{(p_1,q_1)}(T_1, J)$ and $S_2 = \text{UBS}_{(p_2,q_2)}(T_2, J)$. Then, we can define an unbiased estimator for the above query, $E_{\text{count}} = |T_1 \bowtie_J T_2|$, using S_1 and S_2 as follows. Observe that given any pair of tuples $t_1 \in T_1$ and $t_2 \in T_2$, where $t_1.J = t_2.J$, the probability that (t_1, t_2) enters $S_1 \bowtie S_2$ is $p_{\min} q_1 q_2$, where $p_{\min} = \min\{p_1, p_2\}$. Hence, the following is an unbiased estimator for E_{count}.

$$\hat{J}_{\text{count}}(p_1, q_1, p_2, q_2, S_1, S_2) = \frac{1}{p_{\min} q_1 q_2} |S_1 \bowtie S_2|. \tag{4.1}$$

When the arguments $p_1, q_1, p_2, q_2, S_1, S_2$ are clear from the context, we omit these arguments and simply write $\hat{J}_{\text{count}}$.

Definition 8 (Join Size Estimation Problem)**.** *Consider a perfect hash function h. Then, given sampling budgets ϵ_1, ϵ_2, our goal is to find parameters (p_1, q_1) and (p_2, q_2) that minimize the variance of $\hat{J}_{count}$ subject to $p_1 q_1 = \epsilon_1$ and $p_2 q_2 = \epsilon_2$. We call (p_1, q_1) and (p_2, q_2) the optimal UBS sampling parameters for count.*

The next theorem provides the variance of $\hat{J}_{\text{count}}$.

[5] We focus on two servers, but the math can easily be generalized to decentralized networks of multiple servers.

Theorem 9. *Let $S_1 = UBS_{p_1,q_1}(T_1, J)$ and*
$S_2 = UBS_{p_2,q_2}(T_2, J)$. The variance of $\hat{J}_{count}$ is as follows:

$$\begin{aligned}\mathrm{Var}(\hat{J}_{count}) &= \frac{1-p}{p}\sum_v a_v^2 b_v^2 + \frac{1-q_2}{pq_2}\sum_v a_v^2 b_v \\ &+ \frac{1-q_1}{pq_1}\sum_v a_v b_v^2 + \frac{(1-q_1)(1-q_2)}{pq_1q_2}\sum_v a_v b_v.\end{aligned}$$

To minimize $\mathrm{Var}(\hat{J}_{\text{count}})$ under a fixed sampling budget, the two tables should always use the same *universe* sampling rate. If $p_1 > p_2$, the *effective universe sampling rate* is only p_2, i.e., only p_2 fraction of the join keys inside T_1 end up in the join of the sample, and the remaining $p_1 - p_2$ fraction is simply *wasted*. Then, we can change the universe sampling rate of T_1 to p_2 and increase the corresponding uniform sampling rate to obtain a better bound on variance.

Lemma 10. *Given tables T_1, T_2 joined on column(s) J, a fixed sampling parameter (p_1, q_1) for T_1, and a fixed effective sampling rate ϵ_2 for T_2, the variance of $\hat{J}_{count}$ is minimized when T_2 uses p_1 as its universe sampling rate and correspondingly ϵ_2/p_1 as its uniform sampling rate.*

Note that Lemma 10 applies to both centralized and decentralized settings, i.e., it applies to any feasible sampling parameter (p_1, q_1) and (p_2, q_2), regardless of how the sampling parameter is decided. Given the above results, we can obtain the optimal sampling parameters for `COUNT` in centralized and decentralized settings with the following theorems (refer to [137] for detailed proofs and derivations).

Theorem 11. *Let T_1 and T_2 be two tables joined on column(s) J. Let a_v and b_v be the frequency of value v in column(s) J of tables T_1 and T_2, respectively. Given their effective sampling rates ϵ_1 and ϵ_2, the optimal UBS sampling parameters (p_1, q_1) and (p_2, q_2) are given by:*
$p_1 = p_2 = \min\{1, \max\{\epsilon_1, \epsilon_2, \sqrt{\frac{\epsilon_1\epsilon_2\sum_v(a_v^2b_v^2 - a_v^2b_v - a_vb_v^2 + a_vb_v)}{\sum_v a_vb_v}}\}\}$ *and $q_1 = \epsilon_1/p$, $q_2 = \epsilon_2/p$.*

Substituting this into Theorem 9, the resulting variance is only a constant factor of Theorem 2's theoretical limit.

Theorem 12. *Given ϵ_1 and ϵ_2, the optimal UBS parameter (p, q_1) and (p, q_2) for `COUNT` in the decentralized setting are given by*

$$p = \min\{1, \max\{\epsilon_1, \epsilon_2, \sqrt{\epsilon_1\epsilon_2(F_aF_b - F_a - F_b + 1)}\}\}$$

and $q_1 = \epsilon_1/p$, $q_2 = \epsilon_2/p$, $F_a = \max_v a_v$, and $F_b = \max_v b_v$.

4.5.2 Sum on Joins

Let E_{sum} be the output of the following simplified query:

```
select sum(T1.W)
from T1 join T2 on J
```

Let F be the sum of column W in the joined samples $S_1 \bowtie S_2$. Then, the following is an unbiased estimator for E_{sum}:

$$\hat{J}_{\text{sum}} = \frac{1}{p_{\min} q_1 q_2} F \tag{4.2}$$

where $p_{\min} = \min\{p_1, p_2\}$.

Now, we can obtain the optimal sampling parameters for `SUM` in centralized and decentralized settings with the following theorem and algorithm (refer to [137] for detailed proofs and derivations).

Theorem 13 (Optimal Sampling for Centralized Sum)**.** *Given effective sampling rates ϵ_1, ϵ_2, the optimal sampling parameters for `SUM` in a centralized setting are given by*

$$\begin{aligned} p = \min\{1, \max\{&\epsilon_1, \epsilon_2, \\ &(\epsilon_1\epsilon_2(\sum_v (a_v^2\mu_v^2 b_v^2 - a_v^2\mu_v^2 b_v - a_v(\mu_v^2 + \sigma_v^2)b_v^2 \\ &+ a_v(\mu_v^2 + \sigma_v^2)b_v))/(\sum_v a_v(\mu_v^2 + \sigma_v^2)b_v))^{1/2}\}\}. \end{aligned}$$

and $q_i = \epsilon_i / p$ for $i = 1, 2$.

The algorithm that determines the universe sampling rate p for a decentralized setting for `SUM` query is as follows:

1. $v_1 = \arg\max_v a_v^2 \mu^2$ amd $v_2 = \arg\max_v a_v(\mu_v^2 + \sigma_v^2)$
2. If $v_1 = v_2$, return

$$\begin{aligned} p = \min\{1, \max\{&\epsilon_1, \epsilon_2, \\ &(\epsilon_1\epsilon_2(a_{v_1}^2 \mu_{v_1}^2 n_b^2 \\ &- a_{v_1}^2 \mu_{v_1}^2 n_b - a_{v_1}(\mu_{v_1}^2 + \sigma_{v_1}^2) n_b^2 \\ &+ a_{v_1}(\mu_{v_1}^2 + \sigma_{v_1}^2) n_b)/(a_{v_1}(\mu_{v_1}^2 + \sigma_{v_1}^2) n_b))^{1/2}\}\}. \end{aligned}$$

3. Otherwise, for $i = 1, 2$, let

$$
\begin{aligned}
&h_i(p) \\
&= (\frac{1}{\epsilon_2} - \frac{1}{p})a_{v_i}^2 \mu_{v_i}^2 n_b + (\frac{1}{\epsilon_1} - \frac{1}{p})a_{v_i}(\mu_{v_i}^2 + \sigma_{v_i}^2)n_b^2 \\
&+ (\frac{p}{\epsilon_1\epsilon_2} - \frac{1}{\epsilon_1} - \frac{1}{\epsilon_2} + \frac{1}{p})a_{v_i}(\mu_{v_i}^2 + \sigma_{v_i}^2)n_b \\
&+ (\frac{1}{p} - 1)a_{v_i}^2 \mu_{v_i}^2 n_b^2.
\end{aligned}
$$

4. Find the roots p^* of $h_1(p) = h_2(p)$, this can be reduced to solving a quadratic equation since $h_i(p)$ are in the form $A_i p + B_i/p + C_i$. Let p_1, p_2 be the roots.
5. Let p_3 and p_4 be the minimizer of h_1 and h_2, given by:

$$
\begin{aligned}
p_3 = \min\{&1, \max\{\epsilon_1, \epsilon_2, \\
&(\epsilon_1\epsilon_2(a_{v_1}^2 \mu_{v_1}^2 n_b^2 \\
&- a_{v_1}^2 \mu_{v_1}^2 n_b - a_{v_1}(\mu_{v_1}^2 + \sigma_{v_1}^2)n_b^2 \\
&+ a_{v_1}(\mu_{v_1}^2 + \sigma_{v_1}^2)n_b)/(a_{v_1}(\mu_{v_1}^2 + \sigma_{v_1}^2)n_b))^{1/2}\}\}.
\end{aligned}
$$

   And similarly for p_4 where we replace v_1 by v_2.
6. Let $p_5 = \max\{\epsilon_1, \epsilon_2\}$.
7. Compute $j = \arg\min_{i:\epsilon_1,\epsilon_2 \le p_i \le 1}\{\max\{h_1(p_i), h_2(p_i)\}\}$.
8. Return $p = p_j$.

4.5.3 Average on Joins

Let E_{avg} be the output of the following simplified query:

```
select avg(T1.W)
from T1 join T2 on J
```

In general, producing an unbiased estimator for AVG is hard.[6] Instead, we define and analyze the following estimator. Let S and C be the SUM and COUNT of column W in $S_1 \bowtie S_2$. We define our estimator as $\hat{J}_{\text{avg}} = S/C$. There are two advantages over using separate samples to evaluate SUM and COUNT: (1) we can use a larger sample to estimate both queries, and (2) since SUM and COUNT will be positively correlated, the variance of their ratio will be lower. Due to the lack of

[6]The denominator, i.e., the size of the sampled join, can even be zero. Also, the expectation of a random variable's reciprocal is not equal to the reciprocal of its expectation.

a close form expression for the variance of the ratio of two random variables, we use a first order bivariate Taylor expansion to approximate the ratio. We have the following result.

Theorem 14. *Let S and C be random variables denoting the sum and cardinality of the join of two samples produced by applying UBS sampling parameters (p_1, q_1) to T_1 and (p_2, q_2) to T_2. Let $p_{\min} = \min\{p_1, p_2\}$. We have:*

$$\mathrm{Var}[S/C] \approx (\frac{E[S]^2}{E[C]^2})(\frac{\mathrm{Var}[S]}{E[S]^2} - \frac{2\mathrm{Cov}[S,C]}{E[S]E[C]} + \frac{\mathrm{Var}[C]}{\mathrm{E}[C]^2}) \tag{4.3}$$

where

$$\begin{aligned}
\mathrm{E}[S] =& p_{\min} q_1 q_2 \sum_v \mu_v a_v b_v \\
\mathrm{E}[C] =& p_{\min} q_1 q_2 \sum_v a_v b_v \\
\mathrm{Var}[S] =& p_{\min} q_1 q_2 (1-q_2)[q_1 \sum_v a_v^2 \mu_v^2 b_v + q_2 \sum_v a_v(\mu_v^2 + \sigma_v^2) b_v^2 \\
& +(1-q_1) \sum_v a_v(\mu_v^2 + \sigma_v^2) b_v] + p_{\min}(1-p_{\min}) q_1^2 q_2^2 a_v^2 \mu_v^2 b_v^2 \\
\mathrm{Var}[C] =& p_{\min} q_1 q_2 [(1-q_2) \sum_v a_v^2 b_v + (1-q_1) q_2 \sum_v a_v b_v^2 \\
& + (1-q_1)(1-q_2) \sum_v a_v b_v + (1-p_{\min}) q_1 q_2 \sum_v a_v^2 b_v^2] \\
\mathrm{Cov}[S,C] =& p_{\min} q_1 q_2 [(1-q_2) q_1 \sum_v a_v^2 \mu_v b_v + (1-q_1) q_2 \sum_v a_v \mu_v b_v^2 \\
& + (1-q_1)(1-q_2) \sum_v a_v \mu_v b_v + (1-p_{\min}) q_1 q_2 \sum_v a_v^2 \mu_v b_v^2]
\end{aligned}$$

In the centralized setting, where a_v, b_v, μ_v and σ_v values are given for all v, every term in the expression $\frac{E[S]^2}{E[C]^2}(\frac{\mathrm{Var}[S]}{E[S]^2} - 2\frac{\mathrm{Cov}[S,C]}{E[S]E[C]} + \frac{\mathrm{Var}[C]}{\mathrm{E}[C]^2}$ that depends on p is proportional to either p or $1/p$

($E[S]/E[C]$ is independent of p). The terms proportional to $\frac{1}{p}$ are $\frac{1}{p}(A-2B+C)$ where

$$
\begin{aligned}
A =& \frac{\sum_v a_v(\mu_v^2+\sigma_v^2)b_v}{(\sum_v a_v\mu_v b_v)^2} + \frac{a_v^2\mu_v^2 b_v^2}{(\sum_v a_v\mu_v b_v)^2} \\
&- \frac{\sum_v a_v^2\mu_v^2 b_v}{(\sum_v a_v\mu_v b_v)^2} - \frac{\sum_v a_v(\mu_v^2+\sigma_v^2)b_v^2}{(\sum_v a_v\mu_v b_v)^2} \\
B =& \frac{1}{\sum_v a_v b_v} + \frac{\sum_v a_v^2 b_v^2 \mu_v}{(\sum_v a_v b_v)(\sum_v a_v\mu_v b_v)} \\
&- \frac{\sum_v a_v^2\mu_v b_v}{(\sum_v a_v b_v)(\sum_v a_v\mu_v b_v)} - \frac{\sum_v a_v\mu_v b_v^2}{(\sum_v a_v b_v)(\sum_v a_v\mu_v b_v)} \\
C =& \frac{\sum_v a_v b_v}{(\sum_v a_v b_v)^2} + \frac{\sum_v a_v^2 b_v^2}{(\sum_v a_v b_v)^2} - \frac{\sum_v a_v^2 b_v}{(\sum_v a_v b_v)^2} - \frac{\sum_v a_v b_v^2}{(\sum_v a_v b_v)^2}
\end{aligned}
$$

The term proportional to p is pD where:

$$
D = \frac{1}{\epsilon_2\epsilon_2}\left(\frac{\sum_v a_v(\mu_v^2+\sigma_v^2)b_v}{(\sum_v a_v\mu_v b_v)^2} - \frac{2}{\sum_v a_v b_v} + \frac{\sum_v a_v b_v}{(\sum_v a_v b_v)^2}\right).
$$

We can find a p that minimizes $\frac{1}{p}(A-2B+C)+pD$ as follows.

Theorem 15. *In the centralized setting, set* $p^- = \max\{\epsilon_1, \epsilon_2\}$, $p^+ = 1$ *and* $p^* = \min\{1, \max\{\epsilon_1, \epsilon_2, \sqrt{\frac{A-2B+C}{D}}\}\}$. *Then the optimal sampling parameter is given by:*

$$
p = \begin{cases}
p^- & \text{if } A-2B+C \le 0 \text{ and } D > 0 \\
p^+ & \text{if } A-2B+C > 0 \text{ and } D \le 0 \\
p^* & \text{if both } A-2B+C \text{ and } D > 0 \\
\arg\min_{p\in\{p^-,p^+\}} \frac{1}{p}(A-2B+C)+pD. & \text{otherwise}
\end{cases}
$$

Minimizing the *worst case* variance for `AVG` (for the decentralized setting) is much more involved than the average case. In most cases, the objective function (variance) is neither convex nor concave in T_2's frequencies. However, note that every term in Theorem 15 is an inner product $\langle x, y\rangle$, where x and y are two vectors stored on `party1` and `party2`, respectively. Fortunately, inner products can be approximated with transferring a very small amount of information using the AMS sketch [38, 101]. With such a sketch, we can derive an approximate sampling rate without communicating the full frequency statistics.

4.6 Multiple Queries and Filters

Creating a separate sample for each combination of aggregation function, join and aggregation columns, and `WHERE` clause is clearly impractical. In this section, we show how to create a single sample to support multiple queries at the cost of some possible loss of approximation quality. First, we ignore the `WHERE` clause and then show how it can be handled too.

Multiple Queries— Consider k queries, where the i-th query Q_i is defined by an aggregate function agg_i, an aggregate column W_i and a join column J_i. The variance of the estimator $\widehat{J}_{Q_i}$ can always be written as $\widehat{J}_{Q_i} = A_i p + B_i/p + C_i$ for some A_i, B_i, and C_i that depend on agg_i, W_i and J_i. To find a single sampling parameter p for all k queries,[7] we need to solve an optimization problem that minimizes a *weighted average* of the variance functions. Specifically, given the user specified weights $\omega_1, \cdots, \omega_k$,

$$p^* = \arg\min_p \sum_i \omega_i \mathrm{Var}[\widehat{J}_{Q_i}] = \sum_i \omega_i (A_i p + B_i/p + C_i) \tag{4.4}$$

where the optimum value p^* can be calculated as:

$$p^* = \min\{1, \max\{, \epsilon_1, \epsilon_2, \sqrt{(\sum_i \omega_i B_i)/(\sum_i \omega_i A_i)}\}\} \tag{4.5}$$

The choice of ω_i values is up to the user. For example, they can be all set to 1, or to Q_i's relative frequency, importance, or probability of appearance (e.g., based on past workloads).

Known Filters— To incorporate `WHERE` clauses, we can simply regard a query with a filter c on $T_1 \bowtie T_2$ as a query without a filter but on a sub-table that satisfies the filter, namely $T' = \sigma_c(T_1 \bowtie T_2)$.

Unknown Filters with Distributional Information— When the set of columns appearing in the `WHERE` clause can be predicted but the exact constants are unknown, a similar technique can be applied. For example, if an equality constraint $C > x$ is anticipated but x may take on 100 different values, we can *conceptually* treat it as 100 separate queries each with a different value of x in its `WHERE` clause. This reduces our problem to that of sampling for multiple queries without a `WHERE` clause, which we know how to handle using equation (4.5).[8] Here, the weight ω_i can be used to exploit any distributional information that might be available. In general, ω_i should be set to reflect the probability of each possible `WHERE` clause appearing in the future. For example, if there R

[7] Note that q is always ϵ/p.

[8] Note that, even though each query in this case is on a different table, they are all sub-tables of the same original table, and hence their sampling rate p is the same.

possible `WHERE` clauses and are equally likely, we can set $\omega_i = 1/R$, but if popular values in a column are more likely to appear in the filters, we can use the column's histogram to assign ω_i.

Unknown Filters— When there is no information about the columns (or their values) in future filters, we can take a different approach. Since the estimator variance is a monotone function in the frequencies of each join key [137],the larger the frequencies, the larger the variance. This means the worst case variance always happens when the `WHERE` clause selects all tuples from the original table. Hence, in the absence of any distributional information regarding future `WHERE` clauses, we can simply focus on the original query without any filters to minimize our worst case variance.

4.7 Empirical Studies

In this section, we empirically study UBS and SUBS in various scenarios. Our experiments aim to answer the following questions:

(i) How does our optimal sampling compare to other baselines in centralized and decentralized settings? (§4.7.2, §4.7.3)

(ii) How well does our optimal UBS sampling handle join queries with filters? (§4.7.4)

(iii) How does our optimal UBS sampling perform when using a single sample for multiple queries? (§4.7.5)

(iv) How does our optimal SUBS sampling compare to existing stratified sampling strategies? (§4.7.6, §4.7.7)

(v) How much does a decentralized setting reduce the resource consumption and sample creation overhead? (§4.7.8)

4.7.1 Experiment Setup

Hardware and Software— We borrowed a cluster of 18 *c220g5* nodes from CloudLab [24]. Each node was equipped with an Intel Xeon Silver 4114 processor with 10 cores (2.2Ghz each) and 192GB of RAM. We used Impala 2.12.0 as our backend database to store data and execute queries.

Datasets— We used several real-life and synthetic datasets:

1. **`Instacart` [5].** This is a real-world dataset from an online grocery. We used their *orders* and *order_products* tables (3M and 32M tuples, resp.), joined on *order_id*.

2. **`Movielens` [131].** This is a real-world movie rating dataset. We used their *ratings* and *movies* tables (27M and 58K tuples, resp.), joined on *movieid*.

	B_1	B_2	B_3	B_4	B_5	B_6
p	0.010	0.015	0.030	0.333	0.667	1.000
q	1.000	0.667	0.333	0.030	0.015	0.010

Table 4.2: Six UBS baselines, each with different p and q .

3. **TPC-H [13]**. We used a scale factor of 100, and joined the fact and largest dimension tables. Specifically, we joined *l_orderkey* of the *lineitem* table with *o_orderkey* of the *orders* table (600M and 150M tuples, resp.).

4. **Synthetic.** To better control the join key distribution, we also generated several synthetic datasets, where tables T_1 and T_2 each had 100 million tuples and a join column J. T_1 had an additional column W for aggregation, drawn from a power law distribution with range [1, 1000]. We varied the distribution of the join key in each table to be one of uniform, normal, or power law, creating 9 different datasets. The values of column J were integers randomly drawn from [1, 10M] according to the chosen distribution. For normal distribution, we used a truncated distribution with $\sigma{=}10M/5$. For power law, we used $\alpha{=}1.5$ for both J and W. We refer to these datasets by their tables' distributions, namely S{*distribution of* T_1,*distribution of* T_2} (e.g., S{*uniform,uniform*}).

Baselines— We compared our optimal UBS parameters (referred to as OPT) against six baselines. The UBS parameters of these baselines, $B_1, \ldots, B_6$, are listed in Table 4.2. B_1 and B_6 are simply universe and uniform sampling, respectively. $B_2, \ldots, B_5$ represent different hybrid variants of these sampling schemes. Sampling budgets were $\epsilon_1 = \epsilon_2 = 0.01$ unless otherwise specified.

Implementation— We implemented our optimal parameter calculations in Python application. Our sample generation logic read required information, such as table size and join key frequencies, from the database, and then constructed SQL statements to build appropriate samples in the target database. We used Python to compute approximate answers from sample-based queries.

Variance Calculations— We generated $\beta{=}500$ pairs of samples for each experiment, and re-ran the queries on each pair, to calculate the variance of our approximations.

4.7.2 Join Approximation: Centralized Setting

Table 4.3 shows the sampling rates used by OPT for different datasets and aggregate functions in the centralized setting. For Synthetic, the optimal parameters were some mixture of uniform and universe sampling when both tables were only moderately skewed (i.e., uniform or normal distributions) for COUNT and SUM, whereas it reduced to a simple uniform sampling for power law distribution. This is due to the higher probability of missing extremely popular join keys with universe sampling. To the contrary, for AVG, OPT reduced to a simple universe sampling. This

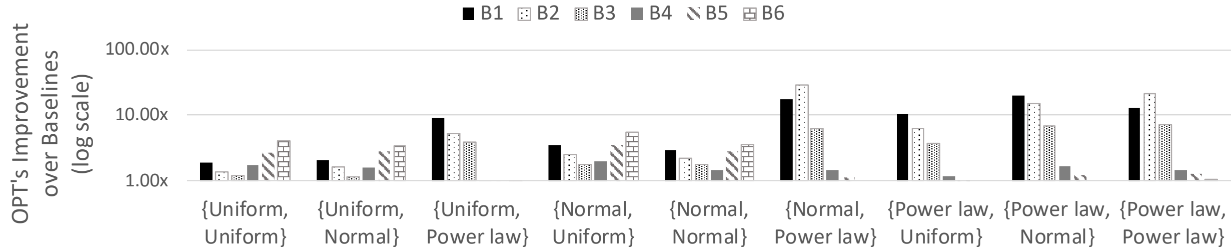


Figure 4.2: OPT's improvement in terms of the estimator's variance for COUNT over six baselines with synthetic dataset.

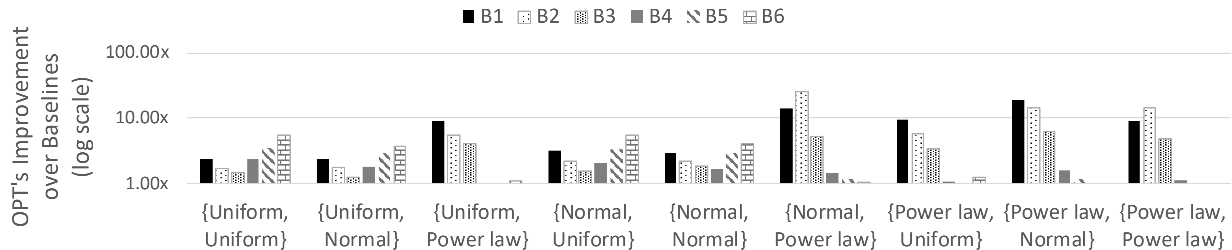


Figure 4.3: OPT's improvement in terms of the estimator's variance for SUM over six baselines with Synthetic dataset.

is because maximizing the output size in this case was the best way to reduce variance. For the other datasets (Instacart, Movielens, and TPC-H), the optimal parameters led to universe sampling, regardless of aggregate type, because the joins in those datasets were PK-FK joins, making uniform sampling less useful for the table with primary keys.

Figure 4.2 shows OPT's improvement over the baselines in terms of variance for COUNT queries. OPT outperformed all baselines in most cases, achieving over 10x lower variance than the worst baseline in several scenarios. Figures 4.3 and 4.4 show the same experiment for SUM and AVG. In both cases, OPT achieved the minimum variance across all sampling strategies, except for AVG when T_1 was a power law distribution. This is because OPT for AVG was calculated from an approximation, rather than a closed-form solution, unlike COUNT or SUM, as discussed in Section 4.5.3. Furthermore, we deliberately randomized both the join key and aggregate columns in Synthetic to create a challenging setting. This, combined with the power law distribution, made it difficult to correctly estimate variances from generated samples. However, UBS with OPT still achieved lowest variance for all aggregates on the real-world datasets Instacart and Movielens, as shown in Figure 4.5. For the selected join key, OPT determined that a full universe sampling was the best sampling scheme, achieving the minimum variance among the baselines.

In summary, this experiment highlights OPT's ability in outperforming simple uniform or universe sampling—or choosing one of them, when optimal—for aggregates on joins.

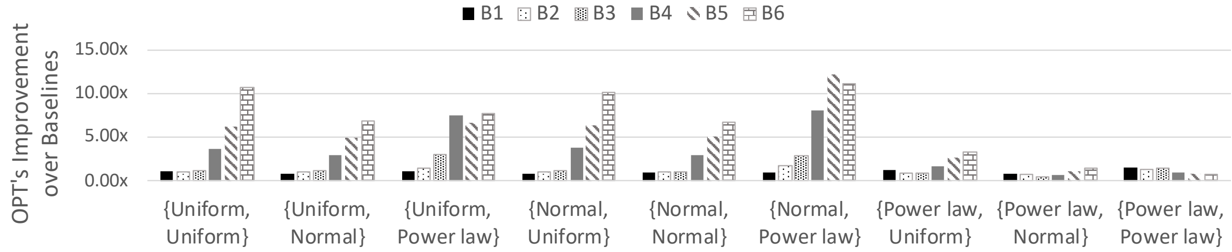
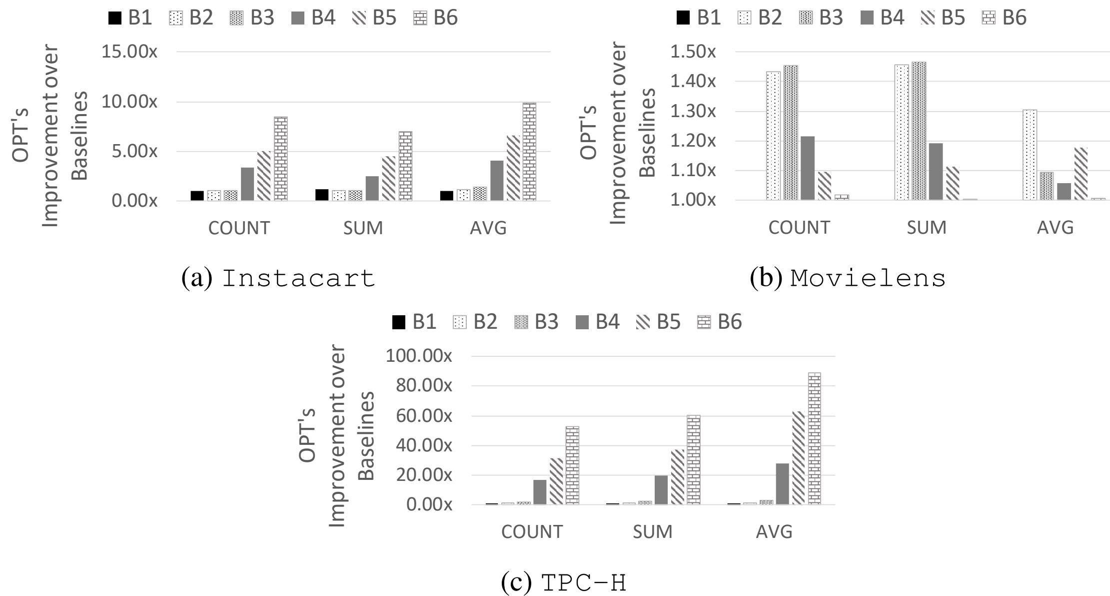


Figure 4.4: OPT's improvement in terms of the estimator's variance for AVG over six baselines with Synthetic dataset.



(a) Instacart

(b) Movielens

(c) TPC-H

Figure 4.5: OPT's improvement in terms of the estimator's variance over six baselines with Instacart, Movielens and TPC-H datasets.

4.7.3 Join Approximation: Decentralized

We evaluated both OPT and other baselines under a decentralized setting using Instacart and Synthetic datasets. Here, we constructed a possible worst case distribution for T_2 that was still somewhat realistic, given the distribution of T_1 and minimal information about T_2 (i..e, T_2's cardinality). To do this, we used the following steps: 1) let $J_{MAX(T_1)}$ be the most frequent join key value in T_1; 2) assign 75% of join key values of T_2 to have the value of $J_{MAX(T_1)}$ and draw the rest of join key values from a uniform distribution.

Figure 4.6 shows the results. For Synthetic, the OPT was the same under both settings whenever there was a power law distribution or the aggregate was AVG. This is because our assumption of the worst case distribution for T_2 was close to a power law distribution. For COUNT and SUM with Synthetic dataset, OPT in the decentralized setting had a much higher variance than OPT in the centralized setting when there was no power law distribution. With Instacart,

Dataset	COUNT		SUM		AVG	
	p	q	p	q	p	q
S{*uniform,uniform*}	0.10	0.10	0.10	0.10	0.01	1.00
S{*uniform,normal*}	0.12	0.08	0.11	0.09	0.01	1.00
S{*uniform,power law*}	1.00	0.01	1.00	0.01	0.01	1.00
S{*normal,uniform*}	0.12	0.08	0.10	0.10	0.01	1.00
S{*normal,normal*}	0.15	0.07	0.13	0.08	0.01	1.00
S{*normal,power law*}	1.00	0.01	1.00	0.01	0.01	1.00
S{*power law,uniform*}	1.00	0.01	1.00	0.01	0.01	1.00
S{*power law,normal*}	1.00	0.01	1.00	0.01	0.01	1.00
S{*power law,power law*}	1.00	0.01	1.00	0.01	0.01	1.00
Instacart	0.01	1.00	0.01	1.00	0.01	1.00
Movielens	0.01	1.00	0.01	1.00	0.01	1.00
TPC-H	0.01	1.00	0.01	1.00	0.01	1.00

Table 4.3: Optimal sampling parameters (p and q) in centralized setting with sampling budget $\epsilon = 0.01$.

Dist. of C	COUNT		SUM		AVG	
	p	q	p	q	p	q
Uniform	0.010	1.000	0.010	1.000	0.010	1.000
Normal	0.018	0.555	0.015	0.648	0.010	1.000
Power law	0.051	0.195	0.050	0.201	0.010	1.000

Table 4.4: Optimal sampling parameters (p and q) for S{*uniform,uniform*} for different distributions of the filtered column C

OPT in the decentralized setting was same as OPT in the centralized setting, which had the minimum variance among the baselines. This illustrates that OPT in the decentralized setting can perform well with real-world data where the joins are mostly PK-FK. This also shows that if a reasonable assumption is possible on the distribution of T_2, OPT can be as effective in the decentralized setting as it is in a centralized one, while requiring significantly less communication.

4.7.4 Join Approximation with Filters

To study OPT's effectiveness in the presence of filters, we used S{*uniform,uniform*} and Instacart datasets. We added an extra column C to T_1 in S{*uniform,uniform*}, with integers values in $[1, 100]$, and tried three distributions (uniform, normal, power law). For Instacart, we used the *order_hour_of_day* column for filtering, which had an almost normal distribution. We used an equality operator ($=$) and chose the comparison value x uniformly at random. We calculated the average variance over all possible values of c.

Table 4.4 shows the sampling rates chosen by OPT, while Figure 4.7 shows OPT's improvement over baselines in terms of average variance. Again, OPT successfully achieved the lowest average

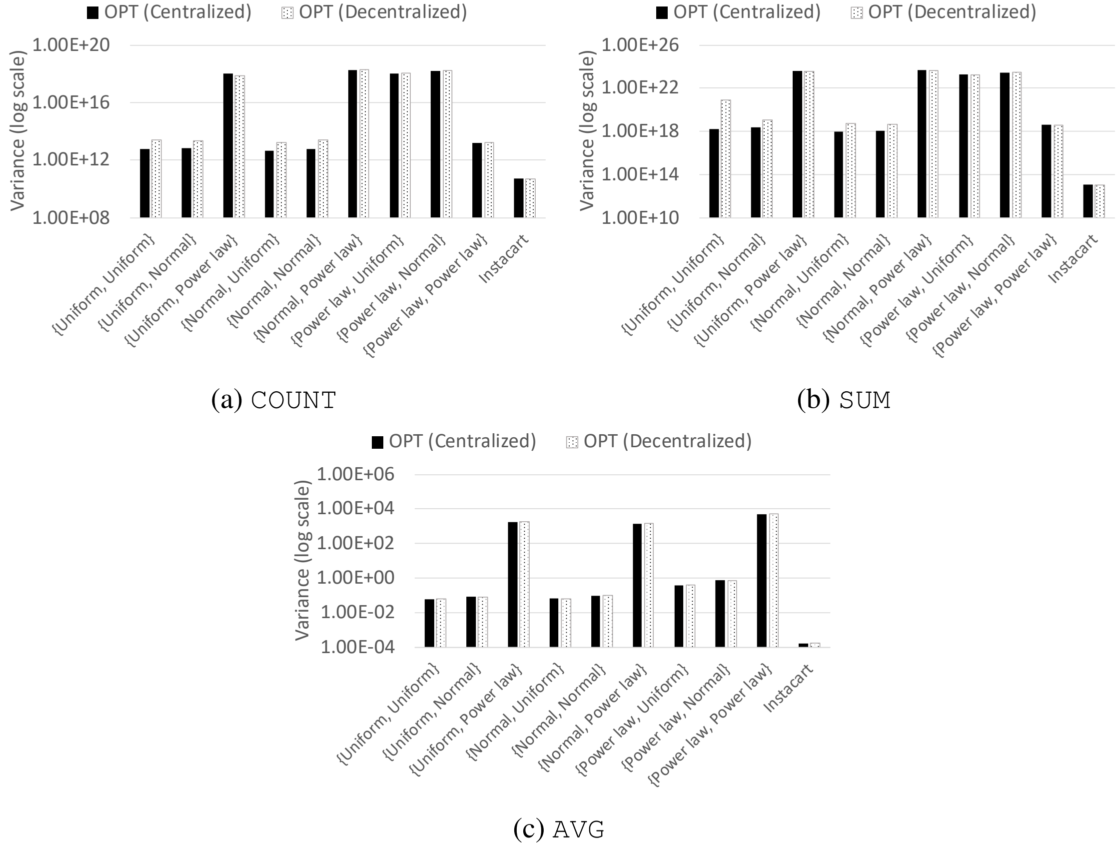


Figure 4.6: Variances of the query estimators for OPT in the centralized and decentralized settings.

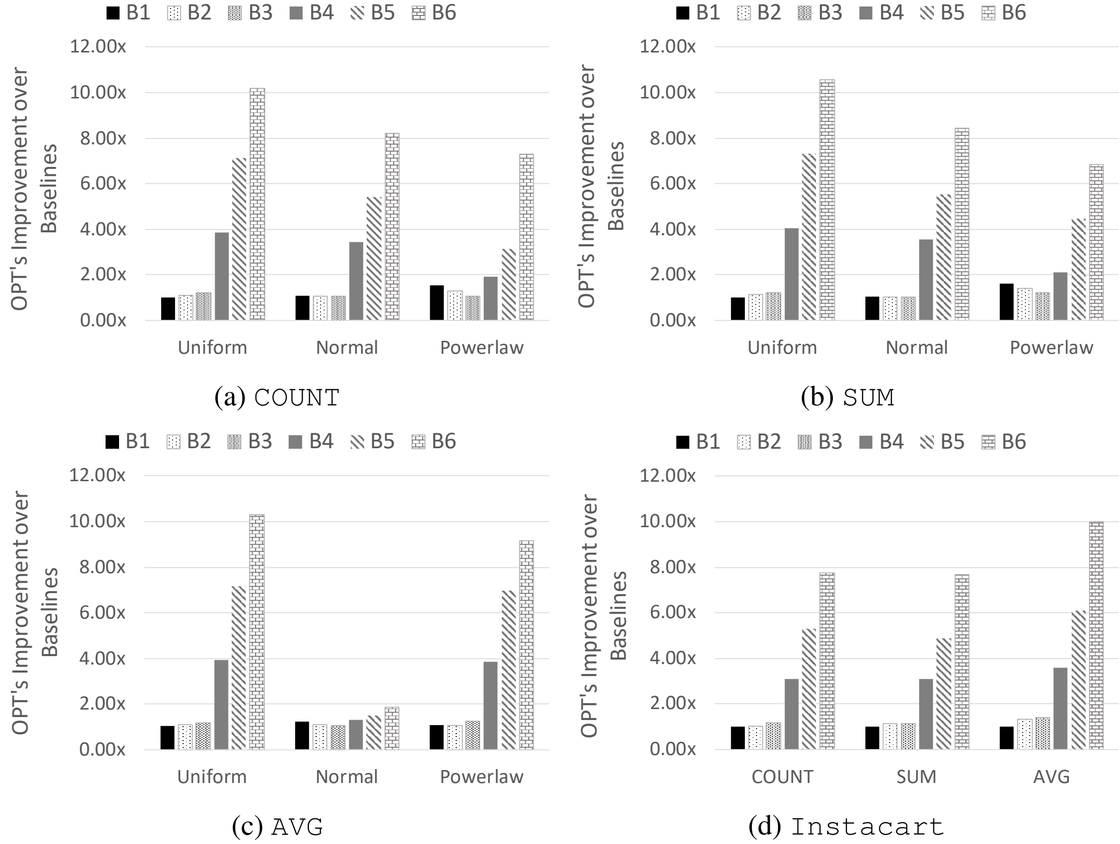
Scheme	COUNT		SUM		AVG	
	p	q	p	q	p	q
OPT (individual)	0.145	0.069	0.125	0.080	0.010	1.000
OPT (combined)	0.133	0.075	0.133	0.075	0.133	0.075

Table 4.5: Sampling parameters (p and q) of OPT using individual samples for different aggregates versus a combined sample ($S\{normal,normal\}$ dataset).

variance among all baselines in all cases, up to 10x improvement compared to the worst baseline. This experiment confirms that UBS with OPT is highly effective for join approximation, even in the presence of filters.

4.7.5 Combining Samples

We evaluated the idea of using a single sample for multiple queries instead of generating individual samples for each query, as discussed in Section 4.6. Here, we use OPT (individual) and OPT (combined) to to denote the use of one-sample-per-query and one-sample-for-multiple-queries, respectively. For OPT (combined), we considered a scenario where each of COUNT, SUM, and AVG is equally likely to appear. Table 4.5 reports the sampling rates chosen in each case. As shown in



(a) COUNT (b) SUM (c) AVG (d) Instacart

Figure 4.7: OPT's improvement in terms of the estimator's variance over six baselines in the presence of filters.

Figure 4.8, without having to generate an individual sample for each query, the variances of OPT (combined) were only slightly higher than those of OPT (individual). This experiment shows that it is possible to create a single sample for multiple queries without sacrificing too much optimality.

4.7.6 Stratified Sampling

We also evaluated SUBS for join queries with group-by. Here, we used the $S\{normal,normal\}$ dataset, and added an extra group column G to T_1 with integers from 0 to 9 drawn from a power law distribution with $\alpha = 1.5$. This time we did not randomize the groups, i.e., G=0 had the most tuples and G=9 had the fewest. This was to study SUBS performance with respect to the different group sizes. As a baseline, we generated stratified samples for T_1 on G with $k_{\text{key}} = 100,000$. and uniform samples for T_2 with a 0.01 sampling budget. We denote this baseline as *SS_UF*. For SUBS, we used parameters that matched the sample size of *SS_UF*, i.e., $k_{\text{key}} = 100, k_{\text{tuple}} = 100,000$. Figure 4.9 shows the variance of query estimators for each of the 10 groups for different aggregations. As expected, SUBS with OPT achieved lower variances than *SS_UF* across all aggregates and groups with different sizes.

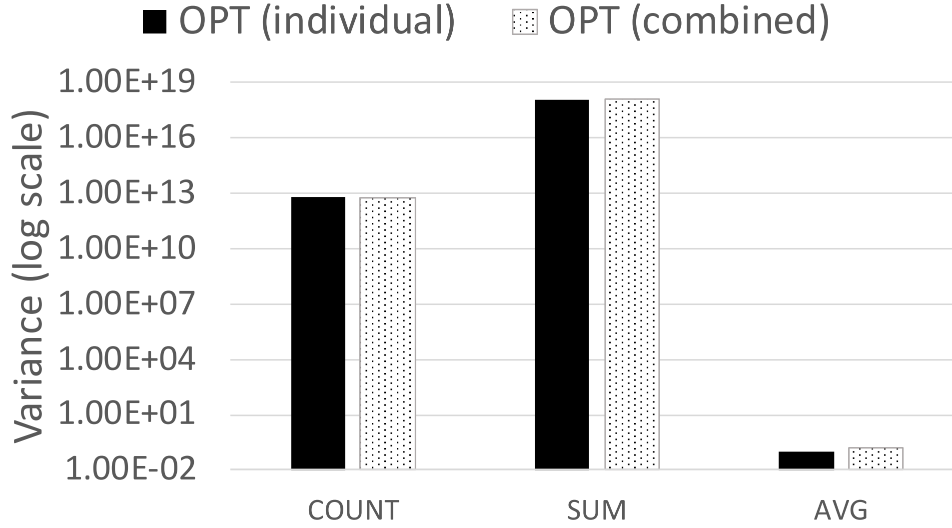
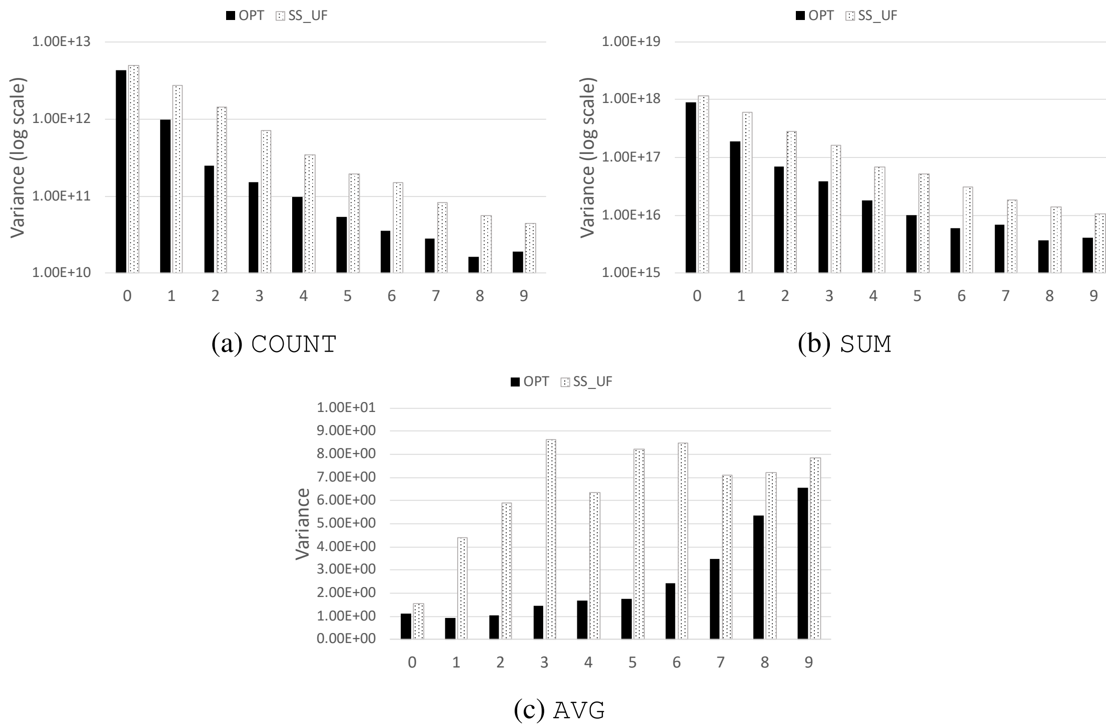


Figure 4.8: Variance of the query estimators for OPT (individual) and OPT (combined) for the $S\{normal,normal\}$ dataset.



(a) COUNT (b) SUM

(c) AVG

Figure 4.9: Query estimator variance per group for for a group-by join aggregate using SUBS versus *SS_UF*.

4.7.7 Optimal Parameters for Stratified Sampling

We empirically studied the effect of different values for two parameters (i.e., k_{key} and k_{tuple}) in SUBS. Similar to Section 4.7.6, we used the $S\{normal,normal\}$ dataset, and added an extra group

column G to T_1 with integers from 0 to 99 drawn from a power law distribution with $\alpha = 2.5$. Larger number of groups and α were used to better observe SUBS's performance with respect to different group sizes and different k_{key} and k_{tuple} values. Again, we did not randomize the groups, i.e., G=0 had the most tuples and G=99 had the fewest. We tried four different values for $k_{\text{key}} \in \{1000, 10000, 100000, 1000000\}$, and k_{tuple} had two possible values: $\{10, 100\}$. We used $epsilon_1 = 0.03$ and $epsilon_2 = 0.01$. We gave a larger budget for T_1 than previous experiments to make all the combinations we tried feasible. We had the total of 8 different combinations of k_{key} and k_{tuple}.

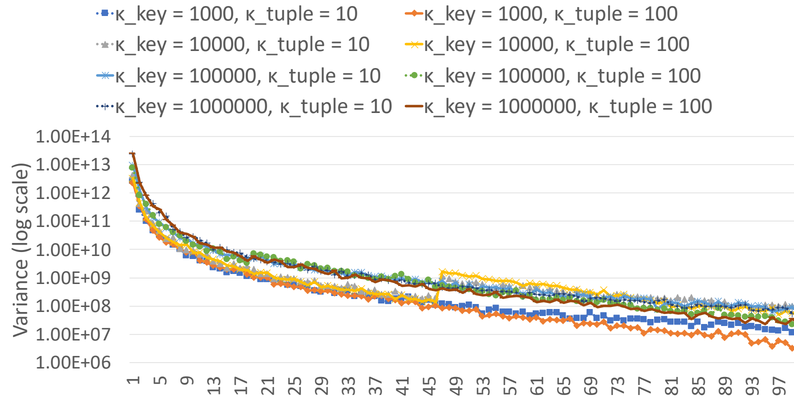
Figure 4.10 shows the variance of query estimators for each of 100 groups with different combinations of k_{key} and k_{tuple} for different aggregations. SUBS achieved the lowest variance overall when $k_{\text{key}} = 1000$. As expected, this was when SUBS was able to categorize every group as *large* and perform full UBS on every group. The variance of the query estimators for a group was increased significantly when the group is categorized as *small* and SUBS was only able to perform uniform sampling on the group. This was evident especially with the combinations that have $k_{\text{key}} = 10000$ as there was a spike in variance when G=47, which was the first *small* group categorized by SUBS with $k_{\text{key}} = 10000$. Unlike k_{key}, k_{tuple} did not have much effect on the variance of query estimators as long as the value was small enough to make the sampling feasible. This experiment shows that SUBS can achieve the lowest variance for all groups when it can categorize every group as *large* and perform full UBS for each group.

4.7.8 Overhead: Centralized vs. Decentralized

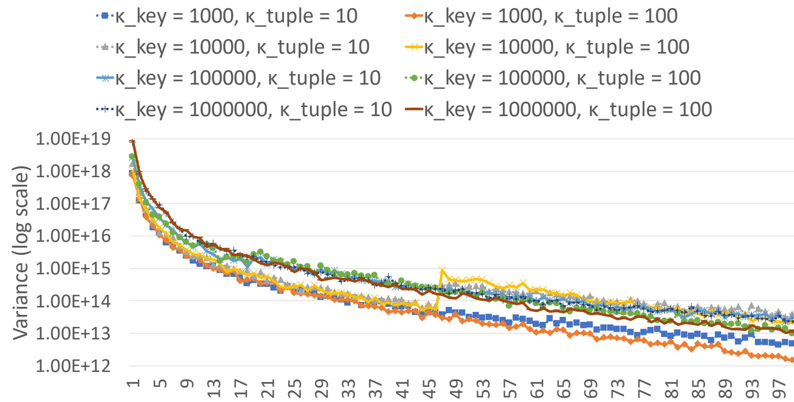
We compared the overhead of `OPT` in centralized versus decentralized settings, in terms of the sample creation time and resources, such as network and disk. `OPT` should have a much higher overhead in the centralized setting, as it requires full frequency information of every join key value in both tables. To quantify their overhead difference, we used `Instacart` and `TPC-H`, and created a pair of samples for `SUM` in each case. Here, the aggregation type did not matter as the time spent calculating p and q was negligible compared to the time taken by transmitting the frequency vectors.

As shown in Figure 4.11, we measured the time for statistics acquisition, sampling rate calculation, and sample table creation. Here, the time taken by collecting the frequencies was the dominant factor. For `Instacart`, it took 65.16 secs from start to finish in the decentralized setting, compared to 99.98 secs in the centralized setting, showing 1.53x improvement in time. For `TPC-H`, it took 59.5 min in decentralized setting, compared to 91.7 mins of the centralized, showing a speedup of 1.54x.

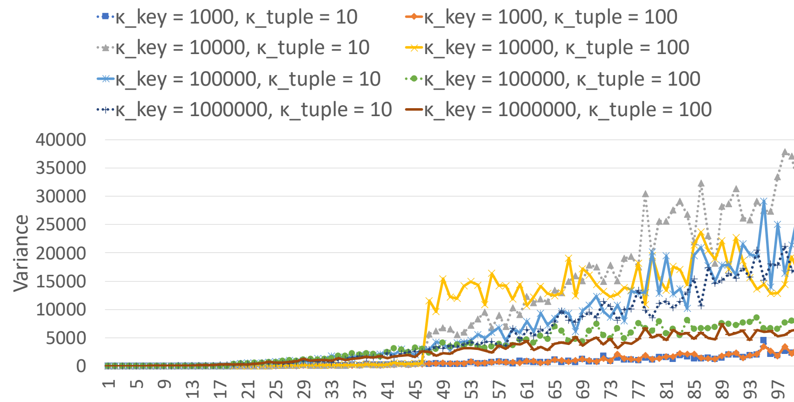
We also measured the total network and disk I/O usage across the entire cluster, as shown



(a) COUNT



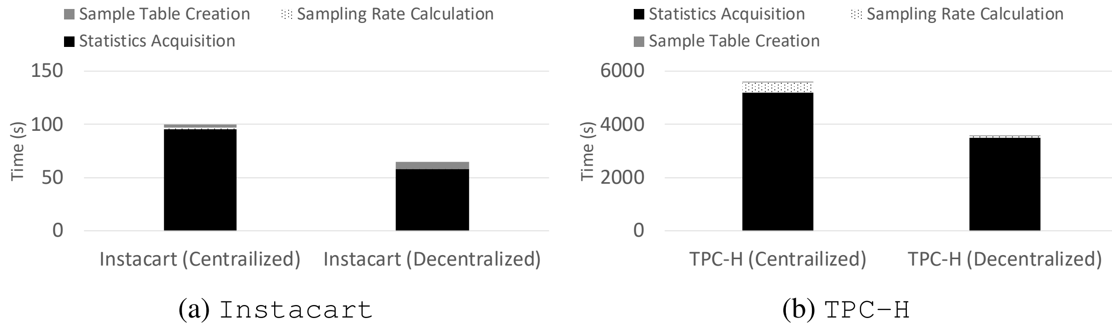
(b) SUM



(c) AVG

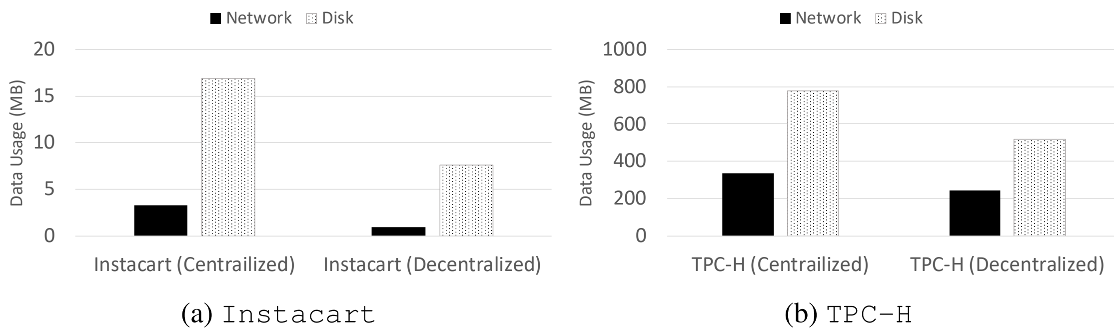
Figure 4.10: Query estimator variance per group for for a group-by join aggregate using SUBS with different values of k_{key} and k_{tuple}.

in Figure 4.12. For `Instacart`, compared to the decentralized setting, the centralized one used 3.66x ($0.9 \rightarrow 3.29$ MB) more network and 2.22x ($7.59 \rightarrow 16.9$ MB) more disk bandwidth. Overall, the overhead was less for `TPC-H`. The centralized in this case used 1.38x ($243.39 \rightarrow 337.04$ MB) more network and 1.49x ($519.03 \rightarrow 776.58$ MB) more disk bandwidth than the decentralized setting.



(a) `Instacart` (b) `TPC-H`

Figure 4.11: Time taken to generate samples for `Instacart` and `TPC-H` in centralized vs. decentralized setting.



(a) `Instacart` (b) `TPC-H`

Figure 4.12: Total network and disk bandwidth used to generate samples for `Instacart` and `TPC-H` in centralized vs. decentralized setting.

This experiment shows the graceful tradeoff between the optimality of sampling and its overhead, making the decentralized variant an attractive choice for large datasets and distributed systems.

4.8 Related Work

Online Sample-based Join Approximation— Ripple Join [125] is an online join algorithm that operates under the assumption that the tuples of the original tables are processed in a random order. Each time, it retrieves a random tuple (or a set of random tuples) from the tables, and then joins the new tuples with the previously read tuples and with each other. A parallel version of Ripple Join [176] is also proposed. SMS join [147] overcomes the problem of the hashed version of Ripple join, which is slow when hash tables exceed memory, using a disk-based sort-merge join. Wander Join [169] tackles the problem of k-way chain join and eliminates the random order requirement of Ripple Join. However, it requires an index on every join column in each of the tables. Using indexes, Wander Join performs a set of random walks and obtains a *non-uniform* but independent sample of the join. Maintaining an approximation of the size of all partial joins can help overcome

the non-uniformity problem [170, 257].

Offline Sample-based Join Approximation— AQUA [115] acknowledges the quadratic reduction and the non-uniformity of the output when joining two uniform random samples. The same authors propose *Join Synopsis* [29], which computes a sample of one of the tables and joins it with the other tables as a sample of the actual join. Chaudhuri et al. [71] also point out that a join of independent samples from two relations does not yield an independent sample of their join, and propose various sampling strategies using precomputed statistics to overcome this problem, but their solution of collecting full frequency information of the relation can be quite expensive. Zhao et al. [257] provide a better trade-off between sampling efficiency and the join size upper bound. *Hashed sampling* [127] is proposed in the context of selectivity estimation for set similarity queries, now known as universe sampling. Block-level uniform sampling [68] is less accurate but more efficient than tuple-level sampling. Bi-level sampling [81, 126] performs Bernoulli sampling at both the block- and tuple-level, as a trade-off between accuracy and I/O cost of sample generation.

AQP Systems on Join— Most AQP systems rely on sampling and support certain types of joins [27, 32, 67, 111, 152, 170, 198, 203]. STRAT [67] discusses the use of uniform and stratified sampling, and how those can support certain types of join queries. More specifically, STRAT only supports PK-FK joins between a fact table and one or more dimension table(s). BlinkDB [32] extends STRAT and considers multiple stratified samples instead of a single one.As previously mentioned, AQUA [27] supports foreign key joins using join synopses. *Icicles* [111] is a new class of samples that includes tuples that are more likely to be required by future queries based on past workloads. These samples, similar to AQUA, only support foreign key joins. PF-OLA [203] is a framework for parallel online aggregation. It studies parallel joins with group-bys, when partitions of the two tables fit in memory. XDB [170] integrates Wander Join in PostgreSQL. Quickr [152] does not create offline samples. Instead, it uses universe sampling to support equi-joins, where the group-by columns and the value of aggregates are not correlated with the join keys. VerdictDB [198] is a universal AQP framework that supports all three types of samples (uniform, universe, and stratified). VerdictDB utilizes a technique called *variational subsampling*, which creates subsamples of the sample such that it only requires a single join—instead of repeatedly joining the subsamples multiple times—to produce accurate aggregate approximations.

Join Cardinality Estimation— There is extensive work on join cardinality estimation (i.e., `COUNT`) in the database community [37, 107, 157, 167, 202, 224, 235, 244] as an important step of the query optimization process for joins. Two-level sampling [79] first samples a set of join values using universe sampling, and then, for each join value sampled, it performs Bernoulli sampling at a rate specific to that value. It differs from the bi-level sampling scheme [126] as it applies two different

sampling methods, whereas bi-level sampling uses only Bernoulli sampling but at different granularity levels. End-biased sampling [107] samples each tuple with a probability proportional to the frequency of its join key value. Index-based sampling is shown to improve cardinality estimation for main-memory databases [167]. Recently, deep learning is utilized to learn inter-table correlations and improve cardinality estimates [157].

Theoretical Studies— The question about the limitation of sample-based approximation of joins, to the best of our knowledge, has not been asked in the theory community. However, the past work in communication complexity on set intersection and inner product estimation has implications for join approximation. In this problem, two parties possess two dimensional vectors x and y and they wish to compute their inner product $t = \langle x, y \rangle$ with as little information exchange as possible. and sends it to Bob, who will in turn estimates $\langle x, y \rangle$ using y and $\beta(x)$. For this problem, [194] shows that any estimator produced by s bits of communication has variance at least $\Omega(dt/s)$. Estimating inner product for $0, 1$ vectors is directly related to estimating `SUM` and `COUNT` for an PK-FK join. A natural question is whether join is still hard even if frequencies are all larger than 1. Further, the question of whether estimating `AVG` is also hard is not answered by prior work.

4.9 Summary

The goal of studying SUBS with its optimal parameters is to improve our understanding of join approximation using offline samples, and formally address some of the key open questions faced by practitioners using and building AQP engines. We discussed generic sampling schemes that cover the most commonly used sampling strategies, as well as as their combinations. Within these schemes, we (1) provided an informational-theoretical lower bound on the lowest error achievable by any offline sampling scheme, (2) derived optimal strategies that match this lower bound within a constant factor, and (3) offered a decentralized variant that requires minimal communication of statistics across the network. Finally, we empirically validated our findings—and the optimality of this sampling scheme—through extensive experiments on multiple datasets.

CHAPTER 5

Distributed Lock Management with RDMA: Decentralization without Starvation

5.1 Motivation

With the advent of high-speed RDMA networks and affordable memory prices, distributed in-memory systems have become increasingly more common [171, 222, 251]. The reason for this rising popularity is simple: many of today's workloads can fit within the memory of a handful of machines, and they can be processed and served over RDMA-enabled networks at a significantly faster rate than with traditional architectures.

A primary challenge in distributed computing is lock management, which forms the backbone of many distributed systems accessing shared resources over the network. Examples include OLTP databases [210,228], distributed file systems [114,218], in-memory storage systems [171,208], and any system that requires synchronization, consensus, or leader election [63, 141, 165]. In a transactional setting, the key responsibility of a lock manager (LM) is ensuring both serializability—or other forms of isolation—and starvation-free behavior of competing transactions [205].

Centralized Lock Managers (CLM)— In traditional distributed databases, each node is in charge of managing the locks for its own objects (i.e., tuples hosted on that node) [17, 22, 132, 160]. In other words, before remote nodes or transactions can read or update a particular object, they must communicate with the LM daemon running on the node hosting that object. Only after the local LM grants the lock can the remote node or transaction proceed to read or modify the object. Although distributed, these LMs are still *centralized*, as each LM instance represents a *central* point of decision for granting locks on the set of objects assigned to it.

Because each CLM instance has *global knowledge* and full visibility into the operations performed on its objects, it can guarantee many strong and desirable properties. For example, it can queue all lock requests and take holistic actions [237, 245], prevent starvation [132, 148, 160], and even employ sophisticated scheduling of incoming requests to bound tail latencies [138, 153, 173,

254].

Unfortunately, the operational simplicity of having global knowledge is not free [117], even with low-latency RDMA networks. First, the CLM itself—specifically, its CPU—becomes a performance bottleneck for applications that require high-throughput locking and unlocking operations and as transactional workloads scale up or out. Second, the CLM becomes a single point of failure [117]. Consequently, many modern distributed systems do not use CLMs in practice; instead, they rely on decentralized lock managers, which offer better scalability and reliability by avoiding a single point of contention and failure [117].

Decentralized Lock Managers (DLM)— To avoid the drawbacks of centralization and to exploit fast RDMA networks, decentralized lock managers are becoming more popular [83, 97, 192, 240]. This is because RDMA operations enable transactions to acquire and release locks on remote machines at extremely low latencies, *without* involving any remote CPUs. In contrast to CLMs, RDMA-based decentralized approaches offer better CPU usage, scalability, and fault tolerance.

Unfortunately, existing RDMA-based DLMs take an extremist approach, where they either completely forgo the benefits of maintaining global knowledge and rely on blind fail-and-retry strategies to achieve higher throughput [83, 240], or they emulate global knowledge using distributed queues and additional network round-trips [192]. The former can cause starvation and thereby higher tail latencies, while the latter significantly lowers throughput, undermining the performance benefits of decentralization.

Challenges— There are two key challenges in designing an efficient DLM. First, to avoid data races, RDMA-based DLMs [83, 240] must only rely on RDMA atomic operations: fetch-and-add (FA) and compare-and-swap (CAS). FA atomically adds a constant to a remote variable and returns the previous value of the variable. CAS atomically compares a constant to the remote variable and updates the variable only if the constant matches the previous value. Although CAS does not always guarantee successful completion, it is easy to reason about, which is why all previous RDMA-based DLMs have relied on CAS for implementing exclusive locks [83, 97, 192, 240]. Consequently, when there is high contention in the system, protocols relying on CAS require multiple retries to acquire a lock. These blind and unbounded retries cause starvation, which increases tail latency. Second, the lack of global knowledge complicates other run-time issues, such as deadlock detection and mitigation.

Our Approach— In this chapter, we propose a fully Decentralized and Starvation-free Lock management (DSLR) algorithm to mitigate the aforementioned challenges. Our key insight is the following: a distributed lock manager can be fully decentralized and yet exchange the partial knowledge necessary for avoiding blind retries, preventing starvation and thereby reducing tail latencies. Specifically, DSLR adapts Lamport's bakery algorithm [164] to a decentralized setting with RDMA

capabilities, which itself poses a new set of interesting challenges:

1. The original bakery algorithm assumes two unbounded counters per object. However, the current RDMA atomic operations are limited to 64-bit words, which must accommodate two counters—for shared and exclusive locks—in order to implement the bakery algorithm, leaving only 16 bits per counter. Because one is forced to use either RDMA CAS or FA, it is difficult to directly and efficiently apply bounded variants of bakery algorithms [145, 227] in an RDMA context.[1]

2. To compete with existing RDMA-based DLMs, our algorithm must be able to acquire locks on uncontented objects using a *single* RDMA atomic operation. However, the original bakery algorithm requires setting a semaphore, reading the existing tickets, and assigning the requester the maximum ticket value.

DSLR overcomes all of these challenges (see §5.4) and, to the best of our knowledge, is the first DLM to extend Lamport's bakery algorithm to an RDMA context.

Contributions— We make the following contributions:

1. We propose a fully decentralized and distributed locking algorithm, DSLR, that extends Lamport's bakery algorithm and combines it with novel RDMA protocols. Not only does DSLR prevent lock starvation, but it also delivers higher throughput and significantly lower tail latencies than previous proposals.

2. DSLR provides fault tolerance for transactions that fail to release their acquired locks or fall into a deadlock. DSLR achieves this goal by utilizing *leases* and determining lease expirations using a locally calculated elapsed time.

3. Through extensive experiments on TPC-C and micro-benchmarks, we show that DSLR outperforms existing RDMA-based LMs; on average, it delivers $1.8\times$ (and up to $2.8\times$) higher throughput, and $2.0\times$ and $18.3\times$ (and up to $2.5\times$ and $47\times$) lower average and 99.9% percentile latencies, respectively.

The rest of this chapter is organized as follows. Section 5.2 provides background material and the motivation behind DSLR. Section 5.3 discusses the design challenges involved in distributed and decentralized locking. Section 5.4 explains DSLR's algorithm and design decisions for overcoming these challenges. Section 5.5 describes how DSLR offers additional features often used by modern database systems. Section 5.7 presents our experimental results, and Section 5.8 discusses related

[1]Existing bounded bakery algorithms only support exclusive locks. Also, they either need additional memory and extra operations [227] or rely on complex arithmetics (e.g., modulo [145]) beyond the simple addition offered by FA.

work.

5.2 Background on RDMA

This section provides the necessary background on modern high-speed networks, particularly RDMA operations, followed by an overview of existing RDMA-based approaches to distributed lock management. Familiar readers can skip Section 5.2.1 and continue reading from Section 5.2.2.

5.2.1 RDMA-Enabled Networks

Remote Direct Memory Access (RDMA) is a networking protocol that provides direct memory access from a host node to the memory of remote nodes, and vice versa. RDMA achieves its high bandwidth and low latency with no CPU overhead by using *zero-copy transfer* and *kernel bypass* [209]. There are several RDMA implementations, including InfiniBand [19], RDMA over Converged Ethernet (RoCE) [15], and iWARP [21].

5.2.1.1 RDMA Verbs and Transport Types

Most RDMA implementations support several operations (verbs) that can broadly be divided into two categories:

1. **Two-Sided Verbs (Verbs with Channel Semantics).** `SEND` and `RECV` verbs have channel semantics, meaning a receiver must publish a `RECV` verb (using RDMA API) prior to a sender sending data via a `SEND` verb. These two verbs are called *two-sided* as they must be matched by both sender and receiver. These verbs use the remote node's CPU, and thus have higher latency and lower throughput than one-sided verbs [183].

2. **One-Sided Verbs (Verbs with Memory Semantics).** Unlike the two-sided verbs, `READ`, `WRITE`, and atomic verbs (`CAS` and `FA`) have memory semantics, meaning they specify a remote address on which to perform data operations. These verbs are *one-sided*, as the remote node's CPU is not aware of the operation. Due to their lack of CPU overhead, one-sided verbs are usually preferred over two-sided verbs [149, 183]. However, the best choice, in terms of which verb to use, always depends on the specific application.

RDMA operations take place over RDMA connections, which are of three transport types: *Reliable Connection (RC)*, *Unreliable Connection (UC)*, and *Unreliable Datagram (UD)*. With RC and UC, two queue pairs need to be *connected* and explicitly communicating with each other.

In this chapter, we focus our scope on Reliable Connection (RC), as it is the only transport type that supports atomic verbs, which we use extensively to achieve fully decentralized lock management in Section 5.4. Next, we focus on atomic verbs in more detail.

5.2.1.2 Atomic Verbs

RDMA provides two types of atomic verbs, compare-and-swap (`CAS`) and fetch-and-add (`FA`). These verbs are performed on 64-bit values. For `CAS`, a requesting process specifies a 64-bit *new value* and a 64-bit *compare value* along with a remote address. The value (i.e., *original value*) at the remote address is compared with the *compare value*; if they are equal, the value at the remote address is swapped with the *new value*. The *original value* is returned to the requesting process. For `FA`, a requesting process specifies a value to be added (i.e., *increment*) to a remote address. The *increment* is added to the 64-bit *original value* at the remote address. Similar to `CAS`, the *original value* is returned to the requesting process.

Atomic verbs have two important characteristics that dictate their usage and system-level design decisions:

- Atomic verbs are guaranteed to never experience data races with other atomic verbs.
- The guarantee does not hold between atomic and non-atomic operations. For example, a data race can still occur between `CAS` and `WRITE` operations [18].

These characteristics effectively restrict how one can mix and match these verbs in their design, which is evident in existing RDMA-based lock management solutions—they all rely heavily on `CAS` [83, 97, 192, 240]. A `CAS` operation will only succeed if its condition is satisfied (i.e., the compare value equals the previous value). This characteristic can lead to unbounded and blind retries, which can severely impact performance and cause starvation. For this reason, our approach avoids the use of `CAS` as much as possible; rather, we primarily rely on `FA` which, unlike `CAS`, is guaranteed to succeed. This also solves the issue of lock starvation, as we explain in Section 5.3.1.

5.2.2 Distributed Lock Managers

5.2.2.1 Traditional Distributed Lock Managers

Before discussing RDMA-based distributed lock managers, we briefly review the architecture of a traditional distributed lock manager [132, 160]. Typically, each node runs a lock manager (LM) instance (or daemon), in charge of managing a *lock table* for the objects stored on that node. Each object (e.g., a tuple in the database) is associated with a *lock object* in the lock table of the node that it is stored on.[2] Before reading or modifying a tuple, a transaction must request a (shared or

[2] For simplicity of presentation, here we assume a single primary copy for each tuple.

exclusive) lock on it. The LM instance running on the local node communicates the transaction's lock request to the LM instance running on the remote node hosting the object [132, 160]. The transaction can proceed only after the corresponding (i.e., remote) LM has granted the lock. The locks are released upon commit/abort in a similar fashion, by going through the remote LM. As discussed in Section 5.1, these traditional LMs are distributed but centralized, in the sense that each LM represents a single point of decision for granting the locks on its associated objects. Next, we discuss distributed and decentralized LMs.

5.2.2.2 RDMA-Based Distributed & Decentralized Lock Managers

Unlike traditional distributed lock managers, in RDMA-based DLMs,[3] a local LM (acting on behalf of a transaction) can directly access lock tables in remote nodes instead of going through the remote DLM instance.[4] While this improves performance in most cases, it also requires new RDMA-aware protocols for lock management [83, 97, 192, 240].

To the best of our knowledge, almost all RDMA-based DLMs use atomic verbs (instead of `SEND/RECV`) for two main reasons. First, the `SEND/RECV` verbs are avoided due to their channel semantics; the solution would be no different than traditional client/server-based solutions with CPU involvement. Second, the `READ/WRITE` verbs cannot be used alone due to their vulnerability to data races, which can jeopardize the consistency of the lock objects themselves. Consequently, previous proposals have all used `CAS` [97, 240], or combined `CAS` with `FA` atomic verbs [83, 192], depending on the data structure used for modeling their lock objects.

Lock Representation and Acquisition— Since RDMA atomic verbs operate on 64-bit values, all RDMA-based DLMs use 64-bit values to represent their lock objects, but with slight variations. For example, Devulapalli et al. [97] implement only exclusive locks and use the entire 64-bit value of a lock object to identify its *owner* (i.e., the transaction currently holding the lock). Others [83, 192] implement both shared and exclusive locks by dividing the 64-bit into two 32-bit regions (Figure 5.1). In these cases, the upper 32-bit region represents the state of an exclusive lock, and DLMs use `CAS` to change this value to record the current [83, 97] or last [192] exclusive lock owner of the object; the lower 32-bit region represents the state of shared locks, and DLMs use `FA` to manipulate this value to count the current number of shared lock owners.

Advisory Locking— Note that RDMA-based DLMs typically use *advisory locking*, meaning participants cooperate and follow/obey a locking protocol without the lock manager enforcing it (i.e.,

[3] Note that we use DLM as an acronym for a Decentralized Lock Manager, rather than a Distributed Lock Manager.

[4] Some combine both architectures by using one-sided RDMA for lock acquisition but relying on additional messages (similar to traditional models) to address lock conflicts [192]. These protocols, however, suffer under contended workloads (see §5.7.2).

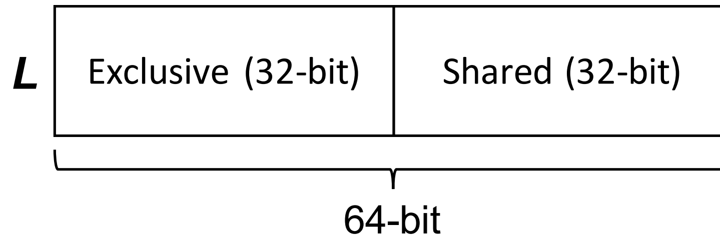


Figure 5.1: The 64-bit representation of a lock object L used by previous RDMA protocols [83, 192].

mandatory locking). This is because one-sided atomic verbs interact with lock tables without involving local DLM instances.

Handling Lock Conflicts— Perhaps the most important aspect of any lock manager is how it handles lock conflicts. There are three possible scenarios of a lock conflict:

(a) **Shared → Exclusive**

(b) **Exclusive → Shared**

(c) **Exclusive → Exclusive**

For example, (a) occurs when a transaction attempts to acquire an exclusive lock on an object that others already have a shared lock on. To handle lock conflicts, DLMs typically use one or a combination of the following mechanisms:

1. *Fail/retry:* a transaction simply fails on conflict and continues trying until the acquisition succeeds [83, 240].

2. *Queue/notify:* lock requests with conflict are enqueued. Once the lock becomes available, a DLM instance or the current lock owner notifies the requester to proceed [97, 192].

Which of these mechanisms is used has important ramifications on the design and performance of a DLM, as discussed next.

5.3 Design Challenges

Since they lack a centralized queue, DLMs face several challenges:

(C1) Lock starvation caused by lack of global knowledge.

(C2) Fault tolerance in case of transaction failures.

(C3) Deadlocks due to high concurrency.

We discuss each of these challenges in the following section.

5.3.1 Lock Starvation

Without a central coordinator, DLMs must rely on their partial knowledge for lock acquisition and handling lock conflicts. Due to their lack of global knowledge, existing RDMA-based DLMs utilize `CAS` and `FA` with blind retries for lock acquisition, which makes them vulnerable to lock starvation.

Lock starvation occurs when a protocol allows newer requests to proceed before the earlier ones, causing the latter to wait indefinitely. The starved transactions might themselves hold locks on other tuples, thus causing other transactions to starve for those locks. Through this cascading lock-wait, lock starvation can cause severe performance degradation and significantly increase tail latencies. Existing DLMs allow for at least one or both of the following types of lock starvation :

(i) **Reader-Writer Starvation:** multiple readers holding shared locks starve a writer from acquiring an exclusive lock.

(ii) **Fast-Slow Starvation:** faster nodes starve slower nodes from acquiring a lock.

5.3.2 Fault Tolerance

DLMs must be able to handle transaction failures (e.g., due to application bugs/crashes, network loss, or node failures), whereby a transaction fails without releasing its acquired locks. As mentioned earlier, RDMA-based DLMs utilize one-sided atomic verbs that do not involve local DLM instances. This makes it difficult for the local DLM to detect and release the unreleased locks on behalf of the failed (remote) transaction. Under *advisory locking*, other transactions will wait indefinitely until the situation is somehow resolved. In several previous RDMA locking protocols [83, 97, 192], a local DLM does not have enough information on its lock table to handle transaction failures. Wei et al. [240] use a *lease* [121] as a fault tolerance mechanism that allows failed transactions to simply expire, allowing subsequent transactions to acquire their locks. However, their approach uses a lease only for shared locks, and cannot handle transactions that fail to release their exclusive locks.

5.3.3 Deadlocks

Deadlocks can happen between different nodes (and their transactions) in any distributed context. However, deadlock detection and resolution can become more difficult without global knowledge

as the number of nodes increases. Furthermore, in some cases, the negative impact of deadlocks on performance can be more severe with faster RDMA networks. This is because one can process many more requests as network throughput increases using RDMA, and assuming a deadlock resolution takes a similar amount of time regardless of network speed, deadlocks can incur a relatively larger penalty on transaction throughput with faster networks.

In the next section, we present our algorithm that overcomes the three aforementioned challenges.

5.4 Our Algorithm

In this section, we present DSLR, a fully decentralized algorithm for distributed lock management using fast RDMA networks. The high-level overview of how DSLR works is shown in Figure 5.2. Based on the challenges outlined in Section 5.3, we start by highlighting the primary design goals of our solution. Next, we describe DSLR in detail, including how it represents locks, handles locking/unlocking operations, and resolves lock conflicts.

5.4.1 Assumptions

In the following discussion, we rely on two assumptions. First, the system clock in each node is well-behaved, meaning none of them advance too quickly or too slowly, and the resolution of system clock in each node is at least ϵ, which is smaller than the maximum lease time that DSLR uses (i.e., 10 ms). This is similar to the assumption in [121], except that we do not require the clocks to be synchronized. Second, the lock manager on each node has prior knowledge of the location of data resources (and their corresponding lock objects) in the cluster. This can be achieved either by using an off-the-shelf directory service for the resources in the cluster (e.g., name server) [92, 109] or by peer-to-peer communications between the lock managers on-the-fly.

5.4.2 Design Criteria

As discussed earlier in Section 5.3, a decentralized DLM faces several challenges, including lock starvation (C1), faults caused by transaction failures (C2), and deadlocks (C3). Next, we explain how our design decisions differ from those of previous DLMs and how they enable us to overcome the aforementioned challenges.

5.4.2.1 Representation of a Lock Object

As mentioned in Section 5.2.2.2, most RDMA-based DLMs split a 64-bit word into two 32-bit regions to represent shared and exclusive locks on an object. Unfortunately, algorithms using this representation rely on the use of CAS, which makes them vulnerable to lock starvation (C1). To solve this problem, we develop a new representation using four 16-bit regions instead of two 32-bit regions as part of our RDMA-based implementation of Lamport's bakery algorithm. We explain the details of our lock object representation in Section 5.4.4.

5.4.2.2 RDMA Verbs

Previous RDMA-based DLMs rely heavily on CAS to change the value of a lock object as they acquire or release a lock. This causes lock starvation (C1) because, as demonstrated in Section 5.3.1, if the value of a lock object keeps changing, a DLM blindly retrying with CAS will continuously fail. Instead of using CAS, DSLR uses FA—which, unlike CAS, is guaranteed to succeed—to acquire and release locks with a single RDMA operation. We describe how we use FA and READ for lock acquisition and handling of lock conflicts in Section 5.4.5 and 5.4.6.

5.4.2.3 Handling Transaction Failures and Deadlocks

To the best of our knowledge, existing RDMA-based DLMs have largely overlooked the issue of transaction failures (C2) and deadlocks (C3). To handle transaction failures, we propose the use of a *lease* [121]. (Note that DrTM [240] also uses a lease for shared locks, but DSLR utilizes it specifically for determining transaction failures. Also, the lease expiration in DSLR is determined locally without the need for synchronized clocks.) We also employ a timeout-based approach to handle deadlocks, utilizing our lease implementation. In addition, we adopt a well-known technique from networking literature, called *random backoffs*, in our bakery algorithm

5.4.3 Lamport's Bakery Algorithm

Before introducing DSLR, we provide a brief background on Lamport's bakery algorithm [164]. Lamport's bakery algorithm is a mutual exclusion algorithm, designed to prevent multiple threads from concurrently accessing a shared resource. In his algorithm, Lamport essentially models a bakery, where each customer entering the bakery receives a ticket with a number that is monotonically increasing. This number is incremented each time a customer enters the bakery. In addition to the ticket numbers, there is also a *global counter* in the bakery, showing the ticket number of the current customer being served, and once the customer is done, this global counter is incremented by 1. The next customer who will be served by the bakery will be the one whose ticket

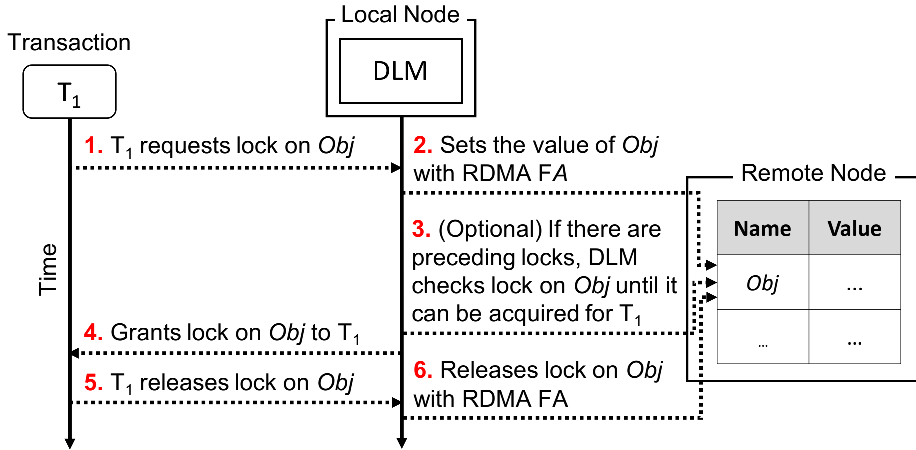


Figure 5.2: A high-level overview of lock acquisition and release in DSLR.

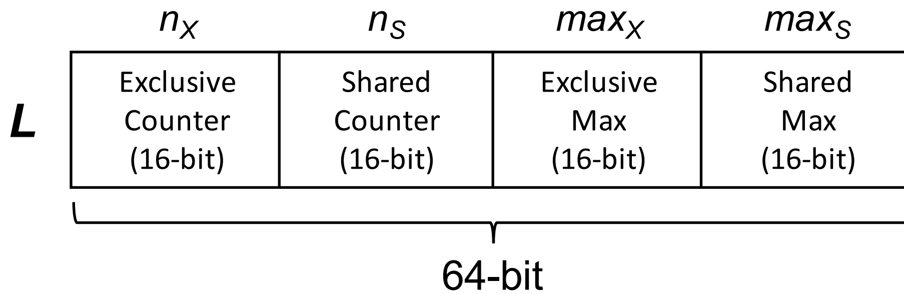


Figure 5.3: DSLR's 64-bit representation of a lock object L.

number matches the current value of the global counter, and so on. In the subsequent sections, we describe how DSLR modifies this original bakery algorithm in an RDMA context with both shared and mutual (i.e., exclusive) locks.

5.4.4 Lock Object Representation

Figure 5.3 shows a 64-bit representation of a lock object that DSLR uses in its RDMA-specific variant of the bakery algorithm, which takes the form of $\{\mathtt{n_X}, \mathtt{n_S}, \mathtt{max_X}, \mathtt{max_S}\}$. Using the bakery analogy from Lamport's original algorithm, the upper 32 bits, $\mathtt{n_X}$ and $\mathtt{n_S}$, are equivalent to global counters, showing the largest ticket numbers of customers (in our case, transactions) that are currently being served for exclusive and shared locks, respectively. The lower 32 bits, $\mathtt{max_X}$ and $\mathtt{max_S}$, are equivalent to the next ticket numbers that an incoming customer will receive for exclusive and shared locks, respectively. By simply incrementing $\mathtt{max_X}$ or $\mathtt{max_S}$ using FA and getting its original value, a transaction obtains a *ticket* with the current max numbers; then it only needs to wait until

the corresponding *counter* value (i.e., n_X or n_S) becomes equal to the number on its obtained ticket.

Note that the original bakery algorithm assumes unbounded counters. However, we are restricted to a 16-bit space (i.e., a maximum of 65,535) to store the value of each counter. The challenge here—if we keep incrementing the values—is an overflow, making the state of in-flight transactions invalid. DSLR circumvents this problem by periodically resetting these counters before one can overflow (see §5.4.9). While doing so, it ensures that all other transactions will wait until the reset is properly done, abiding by the advisory locking rules of DSLR. We will explain how this process works in more detail in subsequent sections.

```
function AcquireLock(tid, L, mode)
    Inputs: tid: the id of the requesting transaction
            L: the requested lock object
            mode: lock mode (shared, exclusive)
    Global : ResetFrom[tid, L]: the value of L that tid needs to use for resetting L
             (accessible by all transactions in the local node)
    Consts : COUNT_MAX = 32,768
    Output: Success or Failure
 1  ResetFrom[tid, L] ← 0
 2  if mode = shared then
 3      prev ← FA (L, max_S, 1)
 4      if prev(max_S) ≥ COUNT_MAX OR prev(max_X) ≥ COUNT_MAX then
 5          FA (L, max_S, -1)
 6          Performs RandomBackoff
 7          if prev(n_S) or prev(n_X) has not changed for longer than twice
              the lease time since last failure then
 8              Reset L // Refer to Lines 11–19 in HandleConflict
 9          return Failure
10      else if prev(max_S) = COUNT_MAX-1 then
11          ResetFrom[tid, L] = {prev(max_X), COUNT_MAX, prev(max_X), COUNT_MAX}
12      if prev(n_X) = prev(max_X) then
13          return Success
14      else
15          return HandleConflict(tid, L, prev, mode)
16  else if mode = exclusive then
17      prev ← FA (L, max_X, 1)
18      if prev(max_S) ≥ COUNT_MAX OR prev(max_X) ≥ COUNT_MAX then
19          FA (L, max_X, -1)
20          Performs RandomBackoff
21          if prev(n_S) or prev(n_X) have not changed for longer than twice
              the lease time since last failure then
22              Reset L // Refer to Lines 11–19 in HandleConflict
23          return Failure
24      else if prev(max_X) = COUNT_MAX-1 then
25          ResetFrom[tid, L] = {COUNT_MAX, prev(max_S),
                                 COUNT_MAX, prev(max_S)}
26      if prev(n_X) = prev(max_X) AND prev(n_S) = prev(max_S) then
27          return Success
28      else
29          return HandleConflict(tid, L, prev, mode)
```

Algorithm 5: The pseudocode for the *AcquireLock* function (see Table 5.1 for procedure definitions).

n_X	n_S	max_X	max_S	Action	Ticket
1	1	1	4		
1	1	2	4	**(a)** FA(L, max_X, 1)	**(b)**{1,1,1,4}
1	1	2	4	**(c)** Waits for n_X = 1 and n_S = 4	

Figure 5.4: An example of a transaction T_3 acquiring an exclusive lock with DSLR on a lock object L.

5.4.5 Lock Acquisition

Our algorithm uses a single FA operation to acquire a lock, or simply queue up for it by adding 1 to $\texttt{max}_\texttt{X}$ or $\texttt{max}_\texttt{S}$ without having to directly communicate with other nodes in the cluster. Algorithm 5 presents the corresponding pseudocode. Figure 5.4 demonstrates an example, where a transaction with tid = 3 (i.e., T_3) wants to acquire an exclusive lock on a lock object L, and L at the time has the value of $\{1, 1, 1, 4\}$. Then, T_3 needs to increment $\texttt{max}_\texttt{X}$ of L by 1 to get a ticket with the next maximum number. This, **(a)** in Figure 5.4, will set L = $\{1, 1, 2, 4\}$ and T_3 will have the ticket, **(b)** in Figure 5.4. For exclusive locks, T_3 needs to wait for both shared and exclusive locks preceding it on L. By comparing the values on its ticket and the current $\texttt{n}_\texttt{X}$ and $\texttt{n}_\texttt{S}$ on L, T_3 knows that there are currently $\texttt{max}_\texttt{S} - \texttt{n}_\texttt{S} = 3$ transactions waiting or holding shared locks on L; thus, T_3 will wait for them, **(c)** in Figure 5.4. Here, the *HandleConflict* function is called subsequently (explained in the next section). Similarly, if T_3 wants to acquire a shared lock on L, it needs to increment $\texttt{max}_\texttt{S}$ and wait until $\texttt{prev}(\texttt{max}_\texttt{X}) = \texttt{n}_\texttt{X}$.

DSLR has an additional logic in place to reset segments of a lock object before they overflow, and to ensure that other transactions take their hands off while one is resetting the value (*Lines 4–9* and *Lines 18–23* in Algorithm 5 for shared and exclusive locks, respectively). This logic enables incoming transactions to reset the counters if necessary, hence preventing situations where they would wait indefinitely for other failed transactions to reset the counters. We describe this resetting procedure in Section 5.4.9.

5.4.6 Handling Lock Conflicts

In our algorithm, a lock conflict occurs when a transaction finds that the current *counter*s of L are less than the unique numbers on its assigned ticket, meaning there are other preceding transactions either holding or awaiting locks on the lock object L. DSLR determines this by examining the return value of FA (i.e., prev) and calling *HandleConflict* if there is a lock conflict. Algorithm 6

Function	Description
val(segment)	represents the value of segment in the 64-bit value val. A segment is one of n_X, n_S, max_X, or max_S.
prev← CAS (L, current, new)	runs CAS on L, changing it from current to new only if the value of L is current. The returned value, prev, contains the original value of L before CAS.
prev← FA (L, segment, val)	runs FA on L, adding val to segment of L. The returned value, prev, contains the original value of L before FA. For example, FA (L, max_X, 1) adds 1 to max_X of L.
val←READ (L)	runs RDMA READ on L and returns its value.

Table 5.1: List of notations and procedures used by DSLR.

Time ↓	L: n_X	L: n_S	L: max_X	L: max_S	T_3: Action	T_3: Ticket
	1	1	1	4		
	1	1	2	4	FA(L, max_X, 1)	{1,1,1,4}
	2	4	2	4	Reset(L)	

Figure 5.5: An example of a transaction T_3 resetting a lock object L to resolve a deadlock.

is the pseudocode for the *HandleConflict* function. Remember that prev is the value of L right before the execution of FA. Here, the algorithm continues polling the value of L until it is tid's turn to proceed with L by comparing the current *counter*s of L with the numbers of its own ticket (*Lines 5–7* for shared, *Lines 8–10* for exclusive locks in Algorithm 6). DSLR detects transaction failures and deadlocks when it still reads the same *counter* values even after twice the length of the proposed lease time has elapsed (*Lines 11–19*). This is determined locally by calculating the time elapsed since the last read from the same *counter* values. This function returns Failure only when transaction failures or deadlocks are detected by DSLR and the counters are already reset. In such a case, the transaction can retry by calling the *AcquireLock* function again. DSLR also avoids busy polling by waiting a certain amount of time proportional to the number of preceding tickets. Specifically, DSLR calculates the sum of the number of preceding exclusive and shared tickets (i.e., wait_count in *Line 20*). Then, it waits for this sum multiplied by a default wait time ω, which can be tuned based on the average RDMA latency of the target infrastructure (5 μs in our cluster). This technique is similar to the dynamic interval polling idea used in [216], which reduces network traffic by preventing unnecessary polling Next, we explain how DSLR handles such failures and deadlocks in the *HandleConflict* function.

```
function HandleConflict(tid, L, prev, mode)
    Inputs: tid: ID of the requesting transaction
        L: the requested lock object
        prev: the value of the lock object at time of FA
        mode: lock mode (shared, exclusive)
    Output: Success or Failure
1  while true do
2      val ← READ (L)
3      if prev(max_X) < val(n_X) or
              prev(max_S) < val(n_S) then
4          return Failure
5      if mode = shared then
6          if prev(max_X) = val(n_X) then
7              return Success
8      else if mode = exclusive then
9          if prev(max_X) = val(n_X) and
             prev(max_S) = val(n_S) then
10             return Success
11     if val(n_X) or val(n_S) have not changed
         for longer than twice the lease time then
12         if mode = shared then
13             reset_val ← {prev(max_X),
               prev(max_S) + 1, val(max_X), val(max_S)}
14         else if mode = exclusive then
15             reset_val ← {prev(max_X) + 1,
               prev(max_S), val(max_X), val(max_S)}
16         if CAS(L, val, reset_val) succeeds then
17             if reset_val(max_X) ≥ COUNT_MAX OR
                 reset_val(max_S) ≥ COUNT_MAX then
18                 Reset L to zero // Refer to Lines 6–7 in
                                   // Algorithm 7
19             return Failure
20     wait_count ← (prev(max_X) − val(n_X)) +
                    (prev(max_S) − val(n_S))
       Wait for (wait_count × ω) μs
```

Algorithm 6: Pseudocode of the *HandleConflict* function (see Table 5.1 for procedure definitions).

5.4.7 Handling Failures and Deadlocks

When DSLR detects transaction failures or deadlocks by checking *counter* values for the duration of the proposed lease time, it calls the *HandleConflict* function to reset the *counter* values of L on behalf of tid (*Line 16*).

Note that, where there is a risk of an overflow (i.e., *counter* reaching COUNT_MAX), tid will also

```
function RELEASELOCK(tid, L, elapsed, mode)
    Inputs: tid: ID of the requesting transaction
        L: the requested lock object
        elapsed: the time elapsed since the lock acquisition
                 of L
        mode: lock mode (shared, exclusive)
    Global : ResetFrom[tid, L]: the value of L that tid needs
             to use for resetting L
             (accessible by all transactions in the local node)
    Output: Success
1  if (elapsed is less than the lease time) or
     (ResetFrom[tid, L] > 0) then
2      if mode = shared then
3          val ← FA (L, n_S, 1)
4      else if mode = exclusive then
5          val ← FA (L, n_X, 1)
6      if ResetFrom[tid, L] ¿ 0 then
7          Repeat CAS(L, ResetFrom[tid, L], 0) until it
               succeeds
           ResetFrom[tid, L] ← 0
8  return Success
```

Algorithm 7: Pseudocode of the *ReleaseLock* function (see Table 5.1 for procedure definitions).

be responsible for resetting. After the reset, tid fails with the lock acquisition. It releases all locks acquired thus far and retries from the beginning. For example, suppose T_3 detects a deadlock and wants to reset L, as shown in Figure 5.5. T_3 basically resets L with CAS such that the next transaction can acquire a lock on L, and this resolves the deadlock. T_3 and other transactions that were waiting on L must retry after the reset. The beauty of this mechanism is that a deadlock will be resolved as long as any waiting transactions (say T_W) reset the *counter* of L, where transactions before T_W can simply retry while transactions after T_W can continue with acquiring locks on L. Note that DSLR detects and handles other types of failures, such as transaction aborts and node failures, using a different mechanism. Specifically, transaction aborts are handled in the same fashion as normal transactions; when a transaction is aborted, DSLR simply releases all its acquired locks. However, to detect node failures (which are less common) or a loss of RDMA connections between the nodes, DSLR relies on sending heartbeat messages regularly (10 seconds by default) between the nodes in the cluster and checking the event status of the message from the RDMA completion queues (CQs). We describe the details of the *ReleaseLock* function in the next section.

Time ↓

L: n_X	n_S	max_X	max_S	T_3: Action	ResetFrom[L,3]
29998	32765	30000	32767		
29998	32765	30000	32768	FA(L, max_S, 1)	{30000,32768,30000,32768}
....					
30000	32768	30000	32768		
0	0	0	0	Reset L to 0	

Figure 5.6: An example of a transaction T_3 resetting a lock object L to avoid overflow of counters.

5.4.8 Lock Release

Releasing a lock object L with DSLR is as simple as incrementing n_X or n_S with FA, unless the lease has already expired. Algorithm 7 is the pseudocode for the *ReleaseLock* function. An extra procedure is only needed if the transaction unlocking the lock object happens to also be responsible for resetting the value of L in order to prevent overflows. This is determined by inspecting the value of ResetFrom[tid, L], which would have been set during *AcquireLock*, if tid is required to perform the resetting of L. In that case, tid will increment *counter*, even if the lease has expired, since it has to reset the value of L. Next, we explain how DSLR resets counters of a lock object to prevent overflows.

5.4.9 Resetting Counters

In DSLR, we have a hard limit of COUNT_MAX, which is $2^{15} = 32,768$, for each 16-bit segment of a lock object L. In other words, DSLR only allows counters to increase until halfway through their available 16-bit space. This is identical to the commonly-used buffer overflow protection technique with a canary value to detect overflows [89]. In our case, DSLR uses the 16th most significant bit as a canary bit to reset the value before it actually overflows.

For example, suppose T_3 wants to acquire an shared lock on L, as shown in Figure 5.6. After performing FA on L, T_3 receives prev = {29998, 32765, 30000, 32767}. Now, max_S of prev is 32,767, which is COUNT_MAX-1, meaning (max_S) of the object L has reached the limit COUNT_MAX. At this point, DSLR waits until T_3 and all preceding transactions are complete by setting ResetFrom[3, L] to {30000, 32768, 30000, 32768}. When T_3 releases its lock on L, DSLR resets the value of L to 0 from ResetFrom[3, L] with CAS. Note that once either max_X or max_S reaches COUNT_MAX, no other transactions can acquire the lock until it is reset. If a transaction detects such a case (i.e., *Line 3 and 17* in Algorithm 5), it reverses the previous FA by decrementing either

$\mathtt{max_X}$ or $\mathtt{max_S}$ and performs a random backoff to help the resetting process. Our use of a random backoff ensures that the repeating CAS will eventually avoid other FAs and reset the counter without falling into an infinite loop. In fact, previous work [161] has formally shown that a network packet can avoid collisions with other packets (and be successfully transmitted over the network) with a bounded number of retries using a random backoff, assuming there is a finite number of nodes in the cluster. Similarly, the expected number of CAS retries to avoid other FAs, and successfully reset the counter, will also be bounded.

Note that the use of a 16-bit space means that, at least in theory, the value of a lock object L can still overflow if there are more than 32,767 transactions trying to acquire a lock on the **same object** at the **same time**. However, this situation is highly unlikely in practice with real-world applications. Nonetheless, we only utilize CAS for resetting counters and avoid redundant CAS calls, unlike previous approaches, since our first FA simultaneously acquires the lock successfully or enqueues for the lock in case of conflicts.

5.5 Supporting Additional Capabilities

In this section, we explain how DSLR supports some additional features that are often needed by modern database systems.

5.5.1 Support for Long-Running Transactions

Mixed workloads are increasingly common [191,207], also known as hybrid transactional/analytical processing (HTAP). The use of a fixed lease time can lead to penalizing long-running queries or transactions. To allow such transactions to complete before their lease expires, we use a *multi-slot leasing* mechanism to support varying lease times. Specifically, to request a longer lease, a transaction can add a number larger than 1 (say k) to the next ticket number (i.e., $\mathtt{max_X}$ or $\mathtt{max_S}$). Here, *Line 3* of Algorithm 5 changes from FA (L, $\mathtt{max_S}$, 1) to FA (L, $\mathtt{max_S}$, k) for shared locks. *Line 17* changes similarly for exclusive locks. Our lease expiration logic is also changed accordingly, whereby each transaction determines its own expiration time based on its own ticket numbers rather than a fixed lease duration for all transactions. In other words, the lease expiration will be proportional to δ, where δ is the difference between the current and the next ticket numbers (i.e., $\delta = (\mathtt{prev}(\mathtt{max_X}) - \mathtt{prev}(\mathtt{n_X})) + (\mathtt{prev}(\mathtt{max_S}) - \mathtt{prev}(\mathtt{n_S}))$). Similarly, the transaction releases its acquired lock by adding k instead of 1 to $\mathtt{n_S}$ or $\mathtt{n_X}$ (i.e., *Lines 3 and 5* of Algorithm 7). Note that DSLR can prevent unfairly long durations by imposing the maximum value of k that can be used by any transaction.

With this multi-slot leasing, a long-running transaction is effectively obtaining multiple tickets with consecutive numbers, while other transactions infer its lease expiration time based on the

L: n_U	n_X	n_S	max_U	max_X	max_S	T_3: Action	Ticket
0	0	0	0	0	2		
0	0	0	1	0	2	(a) FA(L, max_U, 1)	(b) {0,0,0,0,0,2}
0	0	0	1	0	2	(c) Shared lock granted to T_3	
0	0	2	1	0	2	(d) Waits for n_S = 2 for exclusive	

Time ↓

Figure 5.7: An example of a transaction T_3 acquiring an update lock on a lock object `L`.

number of outstanding tickets shown on their own tickets. The maximum lease time possible will be $\phi \times \omega$, where ϕ is the remaining ticket numbers in `L` and ω is the default wait time. This means we can always tune ω to accommodate longer transactions, even when ϕ is small (i.e., there are few remaining tickets). Therefore, multi-slot leasing practically eliminates a hard limit on how long a transaction can remain in the system, thereby allowing long-running transactions to run successfully. We study the effectiveness of this technique in Section 5.7.5.

5.5.2 Lock Upgrades

Many modern databases support lock upgrades. For example, a SQL statement "SELECT ... FOR UPDATE" would acquire shared locks on its target rows to read them first, and then later upgrade those shared locks to exclusive ones after draining the existing shared locks (held by other transactions) so that it can update those rows.

`DSLR` supports lock upgrades by implementing a third type of locks, i.e., *update* locks. An update lock is similar to an exclusive lock, except that a transaction can acquire an update lock even when other transactions already have shared locks on that object. In the presence of those shared locks, the transaction with the update lock (say T_U) can only read the object. Once all other shared locks on that object are released, T_U is finally allowed to write to the object. Other shared and exclusive lock requests that arrive after an update lock has been granted must wait for the update lock to be released.

To implement update locks, we simply introduce two new ticket counters, $\texttt{n}_\texttt{U}$ and $\texttt{max}_\texttt{U}$. This means we must divide our 64-bit lock object `L` between six counters (rather than four). For example, suppose the transaction T_3 wants to acquire an update lock on a lock object `L`, as shown in Figure 5.7. Following the same lock acquisition procedure in Section 5.4.5, T_3 takes its ticket by adding 1 to $\texttt{max}_\texttt{U}$ (**(a)** in Figure 5.7). Even though there are already two other transactions with shared locks on `L` (**(b)** in Figure 5.7), T_3 is still granted a shared lock (**(c)** in Figure 5.7). Once the other two transactions release their shared locks, T_3 is granted an exclusive lock on `L` (**(d)** in Figure 5.7).

5.6 Optimization

In this section, we present a number of optimizations to maximize the performance of our distributed locking algorithm. These optimizations include exponential random backoff and tuning the threshold of our timeout approach.

5.6.1 Exponential Random Backoff

Exponential random backoff is a widely used technique in networking literature (e.g., they are used in the IEEE 802.11 protocol [58]) to resolve network collisions. The idea is to progressively wait longer between retransmission of data in case of a network collision. We found that the performance of DLMs with RDMA degrades significantly when multiple nodes repeatedly conflict on highly contested lock objects and resort to reties/polling. We discovered that adopting the exponential random backoff idea in our distributed locking algorithm can be quite effective at mitigating such scenarios.

In particular, we utilize a binary exponential random backoff. When our algorithm detects a deadlock/timeout by reaching its limit on the maximum number of polls, the lock manager waits W microseconds before retrying the lock request, where W is drawn uniformly at random from the following interval:

$$[0, \max(R \times 2^{c-1}, L)]$$

where R is a default backoff time (100 microseconds in our experiments) and c is the number of consecutive deadlocks/timeouts for the current lock object. To prevent an unlucky node from waiting too long, we also impose a maximum value L so that a node never has to wait too long even when c is large (L=10000 by default).

5.6.2 Tuning the Timeout Threshold

As discussed in Section 5.4.6, our algorithm uses a limit on the maximum number of polls for a lock object (i.e., `max_poll` in Algorithm 5 and 6) as a measure of detecting timeouts and deadlocks. We refer to this parameter as the *timeout threshold* in the rest of the chapter. Considering the fast latency of RDMA `READ` verbs in InfiniBand (i.e., ≈ 8 μs in our experiment setup), one may expect that a large number should be used for this threshold (e.g., 10 or 20 retries) in order to prevent premature timeouts. However, surprisingly, we found that high threshold values perform worse than smaller ones that only allow a handful number of polls. The insight here is that it is significantly more network-efficient for a lock manager to 'fail fast' (and release all its currently held locks in a transactional setting) and try again than to wait longer in hopes that the lock object will eventually become available when in fact there is a deadlock situation. When there is high

contention on popular lock objects in the cluster, the situation quickly exacerbates as there is a cascading effect of every node waiting for a lock object while holding locks on others.

5.7 Evaluation

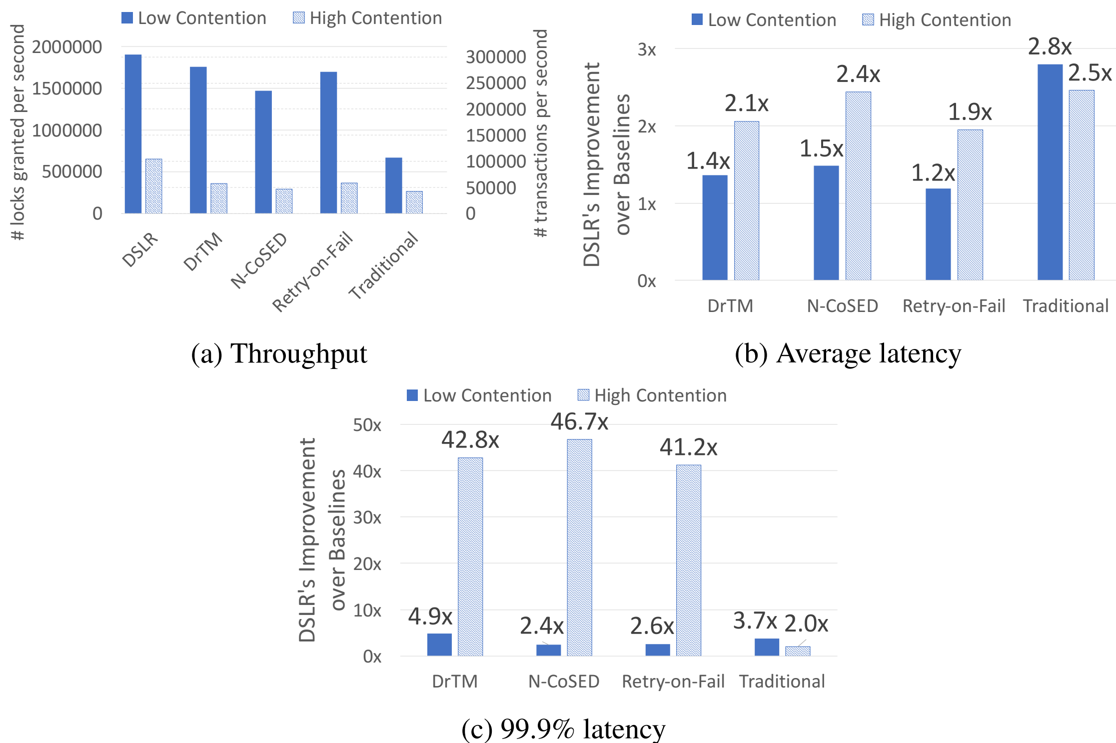
In this section, we empirically evaluate our proposed algorithm, DSLR, and compare it with other RDMA-based approaches to distributed locking. Our experiments aim to answer several questions:

(i) How does DSLR's performance compare against that of existing algorithms? (§5.7.2)

(ii) How does DSLR scale as the number of lock managers increases? (§5.7.3)

(iii) How does DSLR's performance compare against that of queue-based locking in the presence of long-running reads? (§5.7.4)

(iv) How does DSLR support long-running reads effectively with its multi-slot leasing mechanism? (§5.7.5)



(a) Throughput (b) Average latency

(c) 99.9% latency

Figure 5.8: Performance comparison of different distributed lock managers under TPC-C (low and high contention).

5.7.1 Experiment Setup

Hardware— For our experiments, we borrowed a cluster of 32 *r320* nodes from the Apt cluster (part of NSF CloudLab infrastructure for scientific research [16]), each equipped with an Intel Xeon E5-2450 processor with 8 (2.1Ghz) cores, 16GB of Registered DIMMs running at 1600Mhz, and running Ubuntu 16.04 with Mellanox OFED driver 4.1-1.0.2.0. The nodes were connected with ConnectX-3 (1x 56 Gbps InfiniBand ports) via Mellanox MX354A FDR CX3 adapters. The network consisted of two core and seven edge switches (Mellanox SX6036G), where each edge switch was connected with 28 nodes and connected to both core switches with a 3.5:1 blocking factor.

Baselines— For a comparative study of `DSLR`, we implemented the following previous RDMA-based, distributed locking protocols:

1. **`Traditional`** is traditional distributed locking, whereby each node is in charge of managing the locks for its own objects [132, 160]. Although distributed, this approach is still centralized, since each LM instance is a central point of decision for granting locks on the set of objects assigned to it. That is, to acquire a lock on an object, a transaction must communicate with the LM instance in charge of that node. This mechanism uses two-sided RDMA `SEND`/`RECV` verbs with queues.

2. **`DrTM`** [240] is a decentralized algorithm, which uses `CAS` for acquiring both exclusive and shared locks. This protocol implements a *lease* for shared locks, providing a time period for a node to hold the lock. In case of lock conflicts, exclusive locks are retried with `CAS`, and shared locks are retried if the lease has expired. In our experiment, we follow the guidelines provided in their paper for specifying the lease duration.

3. **`Retry-on-Fail`** [83] is another decentralized algorithm, which uses `CAS` for exclusive and `FA` for shared lock acquisition. Their protocol simply retries in all cases of lock conflicts. Although this work is not published, their approach represents an important design choice (i.e., always retry), which merits an empirical evaluation in our experiments. (We refer to this protocol as `Retry-on-Fail`, as it was not named in their report.)

4. **`N-CoSED`** [192] uses `CAS` for exclusive and `FA` for shared lock acquisition. While decentralized, it still tries to obtain global knowledge by relying on distributed queues and extra 'lock request/grant' messages between the DLMs upon lock conflicts.

Implementation— We implemented all baselines in C/C++ (none of them had a readily available implementation). For RDMA, we used *libibverbs* and *librdmacm* libraries with OpenMPI

1.10.2 [20]. Since none of the baselines had a mechanism to handle deadlocks, we also implemented DSLR's timeout-based approach for all of them. For each experiment, we varied the timeout parameter (i.e., maximum number of retries) for each baseline, and only reported their best results.

Servers & Clients— In each experiment, we used one set of machines to serve as LMs and a separate set as clients. Each client machine relied on four worker threads to continuously generate transactions by calling stored procedures on one of the nodes. The LM on that node would then acquire the locks on behalf of the transaction, either locally or on remote nodes. For remote locks, a CLM would contact other CLMs in the cluster, whereas a DLM would acquire remote locks directly. Once all the locks requested by a transaction were acquired, the transaction committed after a *think time* of γ and released all its locks. Unless specified otherwise, we used $\gamma = 0$. The data on each node was entirely cached in memory, while the redo logs were written to disk. Each experiment ran for 5 minutes, which was sufficiently large to observe the steady-state performance of our in-memory prototype.

Workloads— We experimented with two workloads, the well-known TPC-C benchmark and our own microbenchmark For TPC-C, we used two settings: a *low-contention* setting with 10 warehouses per node, and a *high-contention* setting, with one warehouse per node. Each LM instance had a lock table with a lock object for every tuple of its local warehouse(s). Each transaction requested a number of shared or exclusive locks, depending on its type. We used the same proportion of different transaction types as the original TPC-C specification.

5.7.2 Locking Performance For TPC-C

We evaluated the performance of DSLR and all the other baselines by running TPC-C in both low (10 warehouses per node) and high (1 warehouse per node) contention settings. We used a cluster of 16 nodes, each with an LM instance, and we used the remaining 16 machines to generate transactions (see Section 5.7.1). We measured the throughput, average latency, and tail (i.e., 99.9%) latency of the TPC-C transactions under each locking algorithm. As shown in Figure 5.8a, under high contention, our algorithm achieved 1.8–2.5x higher throughput than all other baselines. Under low contention, however, DSLR's throughput was still 2.8x higher than `Traditional`, but was only 1.1–1.3x higher than the other DLMs. This was expected, as all algorithms essentially perform the same operation to acquire an uncontended lock: they all use a single RDMA atomic operation (except for `Traditional`, which still has to use two SEND/RECV operations). Note that SEND/RECV and atomic operations have similar latencies, and the slower performance of `Traditional` is due to its use of two RDMA operations instead of one.

For the same reason, average latencies were also similar for all DLMs under low contention, but were much lower than `Traditional`'s average latency (again, due to the latter's use of two

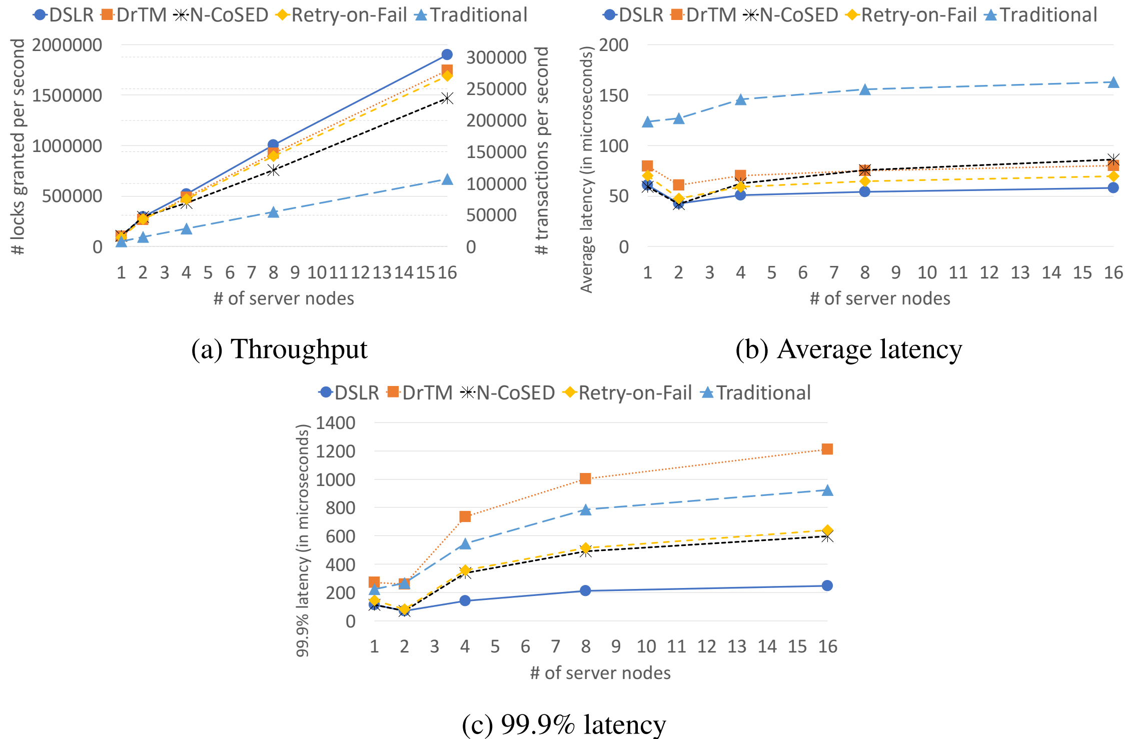


Figure 5.9: Scalability of different distributed locking algorithms with increasing number of nodes.

SEND/RECV verbs instead of a single operation). Figure 5.8b reports the ratio of each baseline's average transaction latency to that of DSLR (i.e., DSLR's speedup). Here, under low contention, DSLR's average latency was 1.2–1.5x lower than other DLMs but 2.8x lower than Traditional. For high contention, however, DSLR's average latency was nearly half of the other techniques, i.e., 1.9–2.5x. This was mainly due to DSLR's utilization of one-sided READ, which is much faster than CAS operations (and blind retries) used by other DLMs in case of lock conflicts.

DSLR's most dramatic improvement was reflected in its tail latencies. As shown in Figure 5.8c, the 99.9 percentile transaction latencies were significantly lower under DSLR than all other baselines: 2.4–4.9x under low contention and up to 46.7x under high contention. This considerable difference underscores the important role of starvation and lack of fairness in causing extremely poor tail performance. Here, Traditional, despite its lower throughput and higher average latency, behaved more gracefully in terms of tail latencies, compared to the other baselines. This was due to Traditional's global knowledge, allowing it to successfully prevent starvation and ensure fairness even in the face of high-contention scenarios. DSLR, on the other hand, achieved the best of both worlds: its decentralized nature allowed for higher throughput, while maintaining sufficient global knowledge allowed it to prevent starvation (and thereby higher tail latencies). The other DLMs that lacked any global knowledge–thus, any mechanism for preventing starvation—skyrocketed in their tail latencies. When compared with Traditional that did not have the issue of lock starvation, the tail latency of DSLR was still about 2x lower. This was because Traditional, as a queue-based

lock manager, still had to use two pairs of SEND/RECV's for each lock/unlock request, one for sending lock/unlock request and another for receiving the response of the lock/unlock request, whereas DSLR only needed a single RDMA operation (i.e., FA) for both locking and unlocking. Overall, the results demonstrate that DSLR is quite robust against lock starvation scenarios and performs better than other baselines in general.

5.7.3 Scalability of DSLR

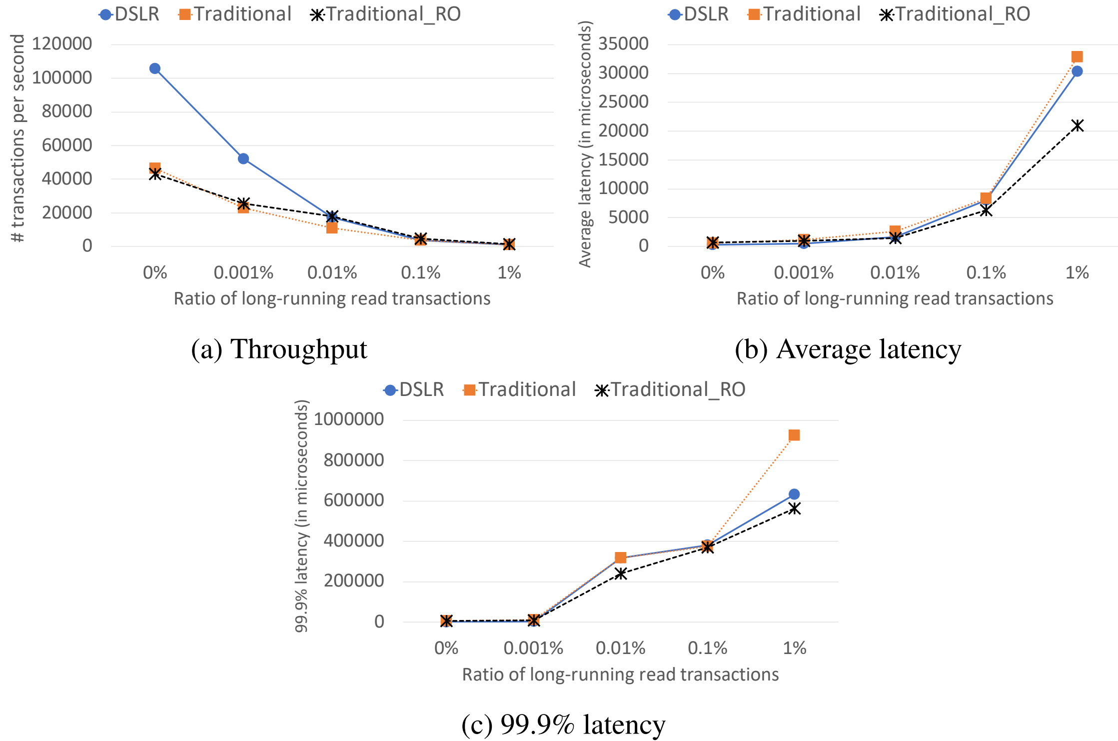
We studied the scalability of DSLR compared to other distributed lock managers. We repeated the experiment with increasing numbers of machines, from 2 to 32. For N machines, $N/2$ were server nodes (and hence lock managers) and the remaining $N/2$ machines generated client transactions. We used the *low-contention* TPC-C setting (10 warehouses per node).

As shown in Figure 5.9, the throughput of all distributed LMs scaled almost linearly (e.g., DSLR achieving 14.5x scalability with 16x additional nodes) as the ratio of the number of server nodes to that of worker threads were constant. N-CoSED and Traditional, which share common characteristics of using queues and SEND/RECV verbs, scaled worse than others in terms of throughput, due to the network congestion caused by their extra messaging. Retry-on-Fail and DrTM scaled better than these queue-based algorithms, as they did not experience as much lock starvation under *low-contention*. However, DrTM showed the worst performance in terms of its tail latency. This was due to its use of lease, as exclusive locks were forced to wait on shared locks until their lease time expired. Overall, DSLR demonstrated a better throughput than other baselines. For average and 99.9% tail latencies, the performance of DSLR remained consistent and was more robust than other baselines even as the number of nodes increased, again thanks to its starvation-free behavior.

5.7.4 Performance with Long-Running Reads

The main advantages of a first-come-first-serve (FCFS) policy are its simplicity, fairness, and starvation-free behavior. However, this also means that an FCFS policy cannot reorder the requests. This can be a drawback in situations where reordering the transactions might improve performance [177], e.g., when there are long-running reads in the system.

To consider such scenarios, we implemented an additional baseline, called **Traditional_RO**, which is similar to Traditional (i.e., queue-based) except that it supports transaction reordering. Specifically, Traditional_RO allows readers ahead of the writers, as long as there is already a shared lock held on the object. To avoid starvation of the writes, we also limited the maximum number of shared locks that can bypass an exclusive lock to 10. This is similar to the strategy proposed in [177], except that we used the number of locks instead of their timestamp to ensure better performance for Traditional_RO. Here, we modified the *high-contention* TPC-C setting de-



(a) Throughput

(b) Average latency

(c) 99.9% latency

Figure 5.10: Performance comparison between DSLR and queue-based (i.e., two-sided) lock managers with/without transaction reordering, under a modified TPC-C with long-running reads.

scribed in Section 5.7.2, as follows: we submitted a long-running read transaction with probability γ and a transaction from the original TPC-C workload with probability $1 - \gamma$. We varied γ exponentially between 0.001% to 1%. Long-running read transactions required a table-level shared lock on $Customer$ table in order to perform a table scan.

Figure 5.10 reports the results for DSLR versus Traditional and Traditional_RO. As expected, the throughput dropped significantly for all lock managers, as soon as long-running reads were introduced, even at 0.001%. The performance of DSLR and Traditional_RO became similar at the ratio of 0.01%, with Traditional_RO starting to perform better at the 0.1% ratio. Traditional_RO's performance was about 1.2–1.4x better than that of DSLR between the ratio of 0.1% and 1%. This is because it began to leverage transaction reordering with enough long-running read transactions. By allowing other table scans and also short reads from $NewOrder$ and $OrderStatus$ transactions ahead of other writes (i.e., $Payment$ and $Delivery$ transactions), Traditional_RO achieved a better performance overall. However, at 1%, the entire system was brought to a halt (only around 1,400 transactions per second for Traditional_RO), compared to when there were no long-running reads (around 110,000 transactions per second for DSLR). This was due to the extreme degree of contention caused by the long-running reads. The experiment demonstrated that queue-based lock managers can benefit from transaction reordering in the presence of long-running reads. However, long-running reads by nature hurt the performance of trans-

actional databases significantly, and there must be a large portion of such long-running reads in the overall workload for transaction reordering to achieve a better performance than DSLR. More importantly, when a shared lock is granted, the lock manager typically does not know when the requesting transaction will release its locks. In other words, the remaining runtime of transaction is not known to the database in general, e.g., the currently held shared lock might be short-lived while the newly arrived one might be long-running. This is perhaps why, to the best of our knowledge, most major databases do not use transaction reordering.

5.7.5 Effectiveness of Our Multi-Slot Leasing

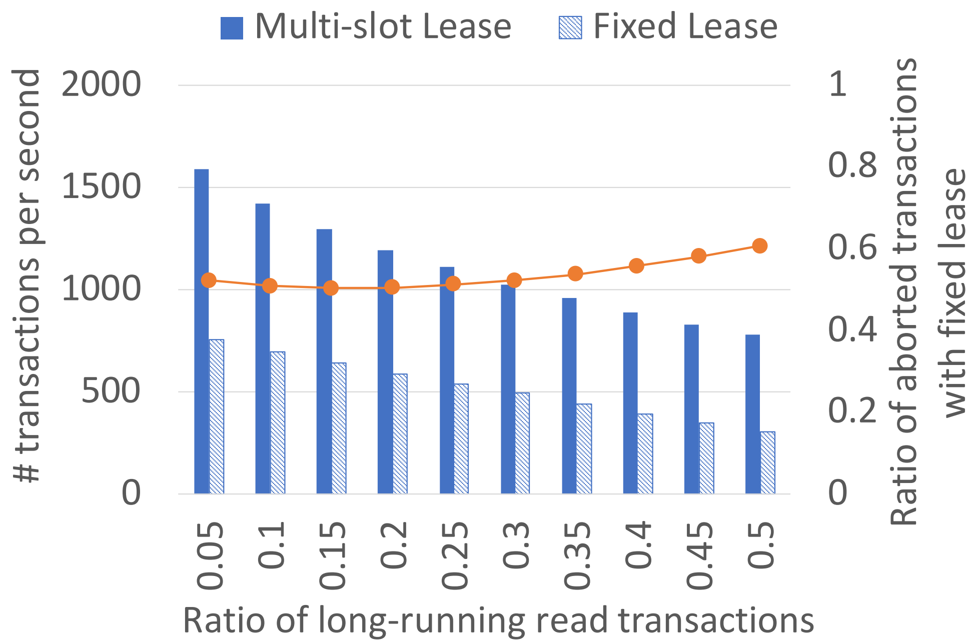


Figure 5.11: Throughput with fixed vs. multi-slot leasing under the modified TPC-C workload.

We studied the effectiveness of our multi-slot leasing mechanism (Section 5.5.1), a technique designed for accommodating long-running transactions. We used the modified TPC-C workload described in Section 5.7.4. For this experiment, we varied the ratio of long-running read transactions from 0.05 to 0.5. We ran DSLR, once with a fixed lease time (i.e., 10 ms) and once with multi-slot leasing. As shown in Figure 5.11, multi-slot leasing led to a better throughput with zero transaction aborts, implying successful execution of the long-running transactions. On the other hand, under DSLR with a fixed lease time, more than half of the transactions aborted. This was caused not only by those long-running transactions that were aborted, but also by other transactions that were blocked by such transactions and eventually timed out and were aborted as well. This confirms that long-running transactions can cause cascading aborts under a fixed lease setting.

The experiment therefore shows the effectiveness of our multi-slot leasing mechanism.

5.8 Related Work

The rise of fast networks has motivated the redesign of distributed systems in general, and databases in particular. While there has been much work on using RDMA for analytics or general data processing [46, 104, 193, 211], here we focus on more relevant lines of work, namely those on distributed lock management and transaction processing. We also discuss other techniques for reducing tail latencies as well the distinction between coordination-free and decentralized protocols.

Distributed Lock Management— Devulapalli et al. [97] propose a distributed queue-based locking using RDMA operations. In their design, each client has its own FIFO (first-in-first-out) queue of waiting clients, to which it will pass the ownership of the current lock. Unlike our algorithm, it requires extra communications using `CAS` among clients in order to enqueue for, and pass the ownership of, a lock. Furthermore, they only support exclusive locks. `N-CoSED` [192] is another RDMA-based distributed locking, where every node uses `CAS` to directly place a lock onto the lock server and exchanges extra "lock request/grant" messages in case of lock conflicts. `N-CoSED` is similar to [97], as each node maintains the list (i.e., a queue) of other nodes waiting for each lock. Chung et al. [83] discuss an alternate and simpler approach by retrying `CAS` repeatedly until the operation is successful for exclusive locks, and continuously checking the exclusive portion of a lock object until it becomes zero for shared locks (i.e., `Retry-on-Fail` in §5.7). Their mechanism is simple and decentralized, yet faces the starvation problem when obtaining exclusive locks on popular objects that have many repeated, continuous reads. In other words, readers starve writers in their model.

RDMA-based Transaction Processing— NAM-DB uses one-sided RDMA read/write and atomic operations to reduce extra communications required by a traditional two-phase commit [54]. However, they only provide snapshot isolation, whereas our proposed distributed lock manager guarantees serializability. Wei et al. design an in-memory transaction processing system, called `DrTM`, that exploits advanced hardware features such as RDMA and Hardware Transactional Memory (HTM) [240]. `DrTM` uses `CAS` to acquire exclusive locks and simply aborts and retries in the case of lock conflicts. FaSST [150] is another system, which utilizes remote procedure calls (RPCs) with two-sided RDMA datagrams to process distributed in-memory transactions. In FaSST, locking is done with `CAS`, relying on aborts and retries upon failure (very similar to [240]). Li et al. propose an abstraction of remote memory as a lightweight file API using RDMA [168], while Dragojević et al. propose distributed platform with strict serializability by leveraging RDMA and non-volatile DRAM [104]. HERD [149] is a key-value store that makes an unconventional decision to use

RDMA writes coupled with polling for communication rather than RDMA reads. They achieve higher throughput by using Unreliable Connection (UC), which unlike Reliable Connection (RC), does not send acknowledgement (ACK/NAK) packets. Their approach is, however, inapplicable to our setting, as all RDMA-based DLMs (including ours) rely heavily on atomic verbs to avoid data races, and RDMA atomic verbs are only available with RC. (Data races can happen with other RDMA verbs.)

Reducing Tail Latencies— A key advantage of DSLR is drastically reducing tail latencies by eliminating starvation. There are other approaches for reducing tail latencies, such as variance-aware transaction scheduling [138, 139], automated explanation [248] or diagnosis [185, 186] of lock-contention problems, redundant computations [112, 236], and choosing indices that are robust against workload changes [188]. All of these approaches are orthogonal to DSLR.

Coordination-free Systems— Bailis et al. [45] show that preserving consistency without coordination is possible when concurrent transactions satisfy a property called *invariant confluence*. Our decentralized algorithm improves the concurrency of distributed systems by allowing a faster coordination in lock management without imposing any extra conditions. In other words, our approach is much more general and does not require that the transactions satisfy invariant confluence.

5.9 Summary

In this chapter, we presented DSLR, an RDMA-based, fully decentralized distributed lock manager that provides a fast and efficient locking mechanism. While existing RDMA-based distributed lock managers abandon the benefits of global knowledge altogether for decentralization, DSLR takes a different approach by adapting Lamport's bakery algorithm and leveraging the characteristics of FA verbs to sidestep the performance drawbacks of the previous CAS-based protocols that suffered from lock starvation and blind retries. DSLR also utilizes the notion of a lease to detect deadlocks and resolve them via its advisory locking rules. Our experiments demonstrate that DSLR results in higher throughput and dramatically lower tail latencies than any existing RDMA-based DLM.

CHAPTER 6

Conclusion

Autonomous databases are not an entirely new concept within the database community, but have recently been in a greater spotlight due to the proliferation of cloud databases. Autonomous databases can significantly reduce database operating costs and free DBAs from less productive tasks like, monitoring and tuning, enabling them to work on more valuable tasks that still require human experts' insights, such as data modeling. In this dissertation, we examined two different aspects of database autonomy—*autonomy from human supervision* and *autonomy among database components*—and proposed new algorithms and frameworks in the three specific areas of database research—*automated tuning*, *performance diagnosis*, and *resource decentralization*—that push boundaries in these two aspects of database autonomy.

Autonomy from human supervision— Ideally, all database management tasks would be fully autonomous, reducing operating costs and improving performance. In reality, however, only a few simple tasks (such as backups and monitoring) are currently considered fully autonomous. More complex tasks still require human supervision, to some degree.

In the context of autonomous databases, there has been much focus on automating database parameter tuning and index creation for optimal database performance. Recent works have shown high levels of autonomy (i.e., between levels of *conditional* and *high*) can be achieved in these areas, but there is still much room for improvement in other areas that have received less attention. In this dissertation, we have concentrated on database management tasks that are mostly overlooked in the context of autonomous databases.

First, we studied a robust physical designer, `CliffGuard`, which is more resilient against workload changes and noisy environments than existing nominal physical designers. `CliffGuard` applies robust optimization (RO) theory in solving physical design problems. Physical designs created by CliffGuard do not need to be re-tuned or re-designed as often as other existing designers, while reducing the average latency of queries up to 14x compared to a state-of-the-art commercial designer. `CliffGuard` can significantly reduce the operating costs incurred from regenerating physical designs, while ensuring highly consistent performance.

Second, we tackled the problem of joins on samples in the context of approximate query processing (AQP), which are considered challenging to solve due to limitations with existing sampling techniques. We proposed a hybrid sampling scheme, called UBS, and derived formulas to calculate its optimal sampling parameters (i.e., OPT). UBS with OPT has been empirically studied through an extensive set of experiments. We showed that it can provide a more accurate query approximation than existing sampling schemes (i.e., uniform or universe sampling) in various scenarios.

Last, we have proposed a new performance explanation framework called DBSherlock for performance diagnosis. Surprisingly, database performance diagnosis is still primarily done manually by DBAs. DBSherlock utilizes techniques from outlier detection and causality analysis to automate this tedious process of performance diagnosis. We have demonstrated that exploratory predicates generated by DBSherlock achieve up to 55% higher F1-measures than those generated by previous techniques.

Overall, we have different database management tasks surrounding automated tuning and performance diagnosis. We believe that each of these works can help to elevate the level of autonomy from human supervision in database systems.

Autonomy among database components— For a fully autonomous database system, we want its components to operate independently of others. Such autonomy among database components is achievable via resource decentralization, but unfortunately, lock management has been difficult to decentralize. This is because lock management operates at the record-level rather than the application-level, unlike other resources, such as compute and storage. Because of this, we have focused on decentralizing lock managers in this dissertation.

We have proposed a fully decentralized and distributed lock manager, called DSLR, that utilizes RDMA (i.e., remote direct memory access) networks. Existing RDMA-based decentralized lock managers all suffer from the problem of starvation, which causes a high tail latency. DSLR solves this problem by adapting traditional Lamport's bakery algorithm, and achieves up to 2.5x higher throughput and up to 46.7x lower tail latency than any other existing RDMA-based decentralized lock managers. We believe DSLR can drastically improve autonomy among database components by fully decentralizing lock management in distributed database systems.

Future work— In this dissertation, we investigated different database management tasks for building a more autonomous database. However, our contributions are independent from one another. these works have been done separately from each other. As we discussed previously, our ultimate goal is to have all database management tasks fully autonomous. Now, it is necessary to merge and integrate them into a single system in order to build a fully autonomous database. It would be fascinating to build such a system that aims to include fully autonomous components only. We anticipate that a completely new database system design may need to be invented first

as a platform for building a fully autonomous database. We also expect this system to be highly modular and easy to expand as non-autonomous components should be easily exchangeable with their autonomous counterparts.

BIBLIOGRAPHY

[1] The 6 levels of vehicle autonomy explained — synopsys automotive. https://www.synopsys.com/automotive/autonomous-driving-levels.html.

[2] Apache Hive Project. http://hive.apache.org/.

[3] Apache Impala. http://impala.apache.org/.

[4] DBSeer. http://www.dbseer.org.

[5] The instacart online grocery shopping dataset 2017. https://www.instacart.com/datasets/grocery-shopping-2017. Accessed: 2019-07-20.

[6] Microsoft Azure. https://azure.microsoft.com/.

[7] OLTPBenchmark. http://oltpbenchmark.com/.

[8] Oracle self-driving database. https://www.oracle.com/database/autonomous-database.html.

[9] Security on-demand announces acquisition of Infobright analytics & technology assets. https://www.prnewswire.com/news-releases/security-on-demand-announces-acquisition-of-infobright-analytics–technology-assets-300465887.html.

[10] SnappyData Inc. http://snappydata.io.

[11] stress-ng. http://tinyurl.com/pw59xs3.

[12] TPC-E benchmark. http://www.tpc.org/tpce/.

[13] TPC-H Benchmark. http://www.tpc.org/tpch/.

[14] What is autonomous database — oracle. https://www.oracle.com/database/what-is-autonomous-database.html.

[15] Rdma over converged ethernet. http://www.roceinitiative.org/, 2016.

[16] Apt. https://www.aptlab.net/, 2017.

[17] Druid — interactive analytics at scale. http://druid.io/, 2017.

[18] Infiniband architecture specification, release 1.3. https://cw.infinibandta.org/document/dl/7859, 2017.

[19] Infiniband trade association. http://www.infinibandta.org, 2017.

[20] Open mpi: Open source high performance computing. https://www.open-mpi.org/, 2017.

[21] Rdma - iwarp. http://www.chelsio.com/nic/rdma-iwarp/, 2017.

[22] Teradata: Business analytics, hybrid cloud & consulting. http://www.teradata.com/, 2017.

[23] TPC-C benchmark. http://www.tpc.org/tpcc/, 2017.

[24] Cloudlab. https://www.cloudlab.us, 2019.

[25] S. Acharya, P. B. Gibbons, and V. Poosala. Aqua: A fast decision support system using approximate query answers. In *VLDB*, 1999.

[26] S. Acharya, P. B. Gibbons, and V. Poosala. Congressional samples for approximate answering of group-by queries. In *SIGMOD*, May 2000.

[27] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. The aqua approximate query answering system. In *ACM Sigmod Record*, volume 28, pages 574–576. ACM, 1999.

[28] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. Join synopses for approximate query answering. In *SIGMOD*, volume 28, pages 275–286. ACM, 1999.

[29] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. Join synopses for approximate query answering. In *ACM SIGMOD Record*, volume 28, pages 275–286. ACM, 1999.

[30] S. Agarwal, H. Milner, A. Kleiner, A. Talwalkar, M. Jordan, S. Madden, B. Mozafari, and I. Stoica. Knowing when you're wrong: Building fast and reliable approximate query processing systems. In *SIGMOD*, 2014.

[31] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. BlinkDB: queries with bounded errors and bounded response times on very large data. In *EuroSys*, 2013.

[32] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 29–42. ACM, 2013.

[33] S. Agrawal, N. Bruno, S. Chaudhuri, and V. R. Narasayya. Autoadmin: Self-tuning database systems technology. *IEEE Data Eng. Bull.*, 29(3), 2006.

[34] S. Agrawal, S. Chaudhuri, L. Kollar, A. Marathe, V. Narasayya, and M. Syamala. Database tuning advisor for microsoft sql server. In *SIGMOD*, 2005.

[35] S. Agrawal, S. Chaudhuri, and V. Narasayya. Automated selection of materialized views and indexes for sql databases. In *VLDB*, 2000.

[36] S. Agrawal, S. Chaudhuri, and V. Narasayya. Materialized view and index selection tool for microsoft sql server 2000. 2001.

[37] N. Alon, P. B. Gibbons, Y. Matias, and M. Szegedy. Tracking join and self-join sizes in limited storage. *Journal of Computer and System Sciences*, 64(3):719–747, 2002.

[38] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *J. Comput. Syst. Sci.*, 58(1):137–147, 1999.

[39] M. Armbrust and et. al. Piql: Success-tolerant query processing in the cloud. *PVLDB*, 5, 2011.

[40] A. Atserias, M. Grohe, and D. Marx. Size bounds and query plans for relational joins. *SIAM J. Comput.*, 42(4):1737–1767, 2013.

[41] B. Babcock and S. Chaudhuri. Towards a robust query optimizer: A principled and practical approach. In *SIGMOD*, 2005.

[42] B. Babcock, S. Chaudhuri, and G. Das. Dynamic sample selection for approximate query processing. In *VLDB*, 2003.

[43] B. Babcock, S. Chaudhuri, and G. Das. Dynamic sample selection for approximate query processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 539–550. ACM, 2003.

[44] S. Babu. Towards automatic optimization of mapreduce programs. In *SoCC*, 2010.

[45] P. Bailis et al. Coordination avoidance in database systems. 2014.

[46] C. Barthels, S. Loesing, G. Alonso, and D. Kossmann. Rack-scale in-memory join processing using rdma. In *SIGMOD*, 2015.

[47] P. Belknap, B. Dageville, K. Dias, and K. Yagoub. Self-tuning for sql performance in oracle database 11g. In *ICDE*, 2009.

[48] A. Ben-Tal, L. El Ghaoui, and A. Nemirovski. *Robust optimization*. Princeton University Press, 2009.

[49] D. G. Benoit. Automatic diagnosis of performance problems in database management systems. In *ICAC*, 2005.

[50] D. Bertsimas and D. B. Brown. Constrained stochastic lqc: a tractable approach. *Automatic Control, IEEE Transactions on*, 52(10), 2007.

[51] D. Bertsimas and et. al. Theory and applications of robust optimization. *SIAM*, 53, 2011.

[52] D. Bertsimas, O. Nohadani, and K. M. Teo. Robust nonconvex optimization for simulation-based problems. *Operations Research*, 2007.

[53] D. Bertsimas and M. Sim. The price of robustness. *Operations research*, 52(1), 2004.

[54] C. Binnig, A. Crotty, A. Galakatos, T. Kraska, and E. Zamanian. The end of slow networks: it's time for a redesign. 2016.

[55] J. R. Birge and et. al. Improving thin-film manufacturing yield with robust optimization. *Applied optics*, 50, 2011.

[56] N. Borisov, S. Uttamchandani, R. Routray, and A. Singh. Why did my query slow down. In *CIDR*, 2009.

[57] S. P. Boyd and L. Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[58] P. Brenner. A technical tutorial on the ieee 802.11 protocol. *BreezeCom Wireless Communications*, 1997.

[59] D. P. Brown, J. Chaware, and M. Koppuravuri. Index selection in a database system, Mar. 3 2009. US Patent 7,499,907.

[60] D. P. Brown, A. Richards, and D. Galeazzi. Teradata active system management, 2008.

[61] D. P. Brown and P. Sinclair. White paper: Real-time diagnostic tools for teradata's parallel query optimizer. Technical report, Teradata Solutions Group, 2000.

[62] P. Buneman, S. Khanna, and T. Wang-Chiew. Why and where: A characterization of data provenance. In *ICDT*. 2001.

[63] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *USENIX OSDI*, 2006.

[64] L. Cao, Q. Wang, and E. A. Rundensteiner. Interactive outlier exploration in big data streams. *PVLDB*, 7, 2014.

[65] C. Chatfield. *Time-series forecasting*. CRC Press, 2002.

[66] S. Chaudhuri, G. Das, and V. Narasayya. Optimized stratified sampling for approximate query processing. *TODS*, 2007.

[67] S. Chaudhuri, G. Das, and V. Narasayya. Optimized stratified sampling for approximate query processing. *ACM Transactions on Database Systems (TODS)*, 32(2):9, 2007.

[68] S. Chaudhuri, G. Das, and U. Srivastava. Effective use of block-level sampling in statistics estimation. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 287–298. ACM, 2004.

[69] S. Chaudhuri, A. K. Gupta, and V. Narasayya. Compressing sql workloads. In *SIGMOD*, 2002.

[70] S. Chaudhuri, H. Lee, and V. R. Narasayya. Variance aware optimization of parameterized queries. In *SIGMOD*, 2010.

[71] S. Chaudhuri, R. Motwani, and V. R. Narasayya. On random sampling over joins. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA.*, pages 263–274, 1999.

[72] S. Chaudhuri and V. Narasayya. Autoadmin what-if index analysis utility. In *SIGMOD Record*, volume 27, 1998.

[73] S. Chaudhuri and V. Narasayya. Self-tuning database systems: A decade of progress. In *VLDB*, 2007.

[74] S. Chaudhuri, V. Narasayya, M. Datar, et al. Linear programming approach to assigning benefit to database physical design structures, Nov. 21 2006. US Patent 7,139,778.

[75] S. Chaudhuri and V. R. Narasayya. An efficient, cost-driven index selection tool for microsoft sql server. In *VLDB*, volume 97, 1997.

[76] A. N. K. Chen and et. al. Heuristics for selecting robust database structures with dynamic query patterns. *EJOR*, 168, 2006.

[77] S. Chen, A. Ailamaki, M. Athanassoulis, P. B. Gibbons, R. Johnson, I. Pandis, and R. Stoica. Tpc-e vs. tpc-c: Characterizing the new tpc-e benchmark via an i/o comparison study. *ACM SIGMOD Record*, 39(3):5–10, 2011.

[78] X. Chen, M. Sim, and P. Sun. A robust optimization perspective on stochastic programming. *Operations Research*, 55, 2007.

[79] Y. Chen and K. Yi. Two-level sampling for join size estimation. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 759–774, 2017.

[80] J. Cheney, L. Chiticariu, and W.-C. Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 2007.

[81] Y. Cheng, W. Zhao, and F. Rusu. Bi-level online aggregation on raw data. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management*, page 10. ACM, 2017.

[82] F. Chu, J. Y. Halpern, and P. Seshadri. Least expected cost query optimization: An exercise in utility. In *PODS*, 1999.

[83] Y. Chung and E. Zamanian. Using rdma for lock management. *arXiv preprint arXiv:1507.03274*, 2015.

[84] S. O.-R. A. V. S. Committee et al. Taxonomy and definitions for terms related to on-road motor vehicle automated driving systems. *SAE Standard J*, 3016:1–16, 2014.

[85] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *Nsdi*, volume 10, page 20, 2010.

[86] G. F. Cooper. A simple constraint-based algorithm for efficiently mining observational databases for causal relationships. *Data Mining and Knowledge Discovery*, 1(2):203–224, 1997.

[87] G. Cormode, M. Garofalakis, P. J. Haas, and C. Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends in Databases*, 4, 2012.

[88] T. M. Cover and J. A. Thomas. Entropy, relative entropy and mutual information. *Elements of Information Theory*, pages 12–49, 1991.

[89] C. Cowan, F. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings*, 2000.

[90] A. Crotty, A. Galakatos, E. Zgraggen, C. Binnig, and T. Kraska. Vizdom: Interactive analytics through pen and touch. *PVLDB*, 2015.

[91] B. Dageville, D. Das, K. Dias, K. Yagoub, M. Zait, and M. Ziauddin. Automatic sql tuning in oracle 10g. In *VLDB*, 2004.

[92] P. B. Danzig, K. Obraczka, and A. Kumar. An analysis of wide-area name server traffic: a study of the internet domain name system. *ACM SIGCOMM Computer Communication Review*, 1992.

[93] S. Das, M. Grbic, I. Ilic, I. Jovandic, A. Jovanovic, V. R. Narasayya, M. Radulovic, M. Stikic, G. Xu, and S. Chaudhuri. Automatically indexing millions of databases in microsoft azure sql database. In *Proceedings of the 2019 International Conference on Management of Data*, pages 666–679. ACM, 2019.

[94] K. Deb. Geneas: A robust optimal design technique for mechanical component design. 1997.

[95] B. K. Debnath, D. J. Lilja, and M. F. Mokbel. Sard: A statistical approach for ranking database tuning parameters. In *2008 IEEE 24th International Conference on Data Engineering Workshop*, pages 11–18. IEEE, 2008.

[96] A. Deligiannakis, V. Stoumpos, Y. Kotidis, V. Vassalos, and A. Delis. Outlier-aware data aggregation in sensor networks. In *ICDE*, 2008.

[97] A. Devulapalli and P. Wyckoff. Distributed queue-based locking using advanced network features. In *ICPP*, 2005.

[98] K. Dias, M. Ramacher, U. Shaft, V. Venkataramani, and G. Wood. Automatic performance diagnosis & tuning in oracle. In *CIDR*, 2005.

[99] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proceedings of the VLDB Endowment*, 7(4):277–288, 2013.

[100] X. Ding and J. Le. Adaptive projection in column-stores. In *FSKD*, 2011.

[101] A. Dobra, M. N. Garofalakis, J. Gehrke, and R. Rastogi. Processing complex aggregate queries over data streams. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, USA, June 3-6, 2002*, pages 61–72, 2002.

[102] I. Doltsinis and et. al. Robust design of non-linear structures using optimization methods. *Computer methods in applied mechanics and engineering*, 194, 2005.

[103] D. Donjerkovic and R. Ramakrishnan. Probabilistic optimization of top n queries. In *VLDB*, 1999.

[104] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. Farm: fast remote memory. In *USENIX NSDI*, 2014.

[105] S. Duan, S. Babu, and K. Munagala. Fa: A system for automating failure diagnosis. In *ICDE*, 2009.

[106] S. Duan, V. Thummala, and S. Babu. Tuning database configuration parameters with ituned. *Proceedings of the VLDB Endowment*, 2(1):1246–1257, 2009.

[107] C. Estan and J. F. Naughton. End-biased samples for join cardinality estimation. In *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*, page 20, 2006.

[108] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96, pages 226–231, 1996.

[109] S. Fitzgerald et al. A directory service for configuring high-performance distributed computations. In *High Performance Distributed Computing, 1997. Proceedings. The Sixth IEEE International Symposium on*, 1997.

[110] V. Ganti, M.-L. Lee, and R. Ramakrishnan. Icicles: Self-tuning samples for approximate query answering. In *VLDB*, volume 176. Citeseer, 2000.

[111] V. Ganti, M.-L. Lee, and R. Ramakrishnan. Icicles: Self-tuning samples for approximate query answering. In *VLDB*, volume 176. Citeseer, 2000.

[112] K. Gardner, S. Zbarsky, S. Doroudi, M. Harchol-Balter, and E. Hyytia. Reducing latency via redundant requests: Exact analysis. *ACM SIGMETRICS Performance Evaluation Review*, 2015.

[113] K. E. Gebaly and A. Aboulnaga. Robustness in automatic physical database design. In *Advances in database technology*, 2008.

[114] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *ACM SIGOPS operating systems review*, 2003.

[115] P. B. Gibbons, V. Poosala, S. Acharya, Y. Bartal, Y. Matias, S. Muthukrishnan, S. Ramaswamy, and T. Suel. Aqua: System and techniques for approximate query answering. *Bell Labs TR*, 1998.

[116] D. Gmach, S. Krompass, A. Scholz, M. Wimmer, and A. Kemper. Adaptive quality of service management for enterprise services. *ACM Transactions on the Web (TWEB)*, 2(1):8, 2008.

[117] K. Gopalakrishna et al. Untangling cluster management with helix. In *Proceedings of the Third ACM Symposium on Cloud Computing*, 2012.

[118] D. Goyal. Approximate query processing at WalmartLabs. https://fifthelephant.talkfunnel.com/2018/43-approximate-query-processing.

[119] G. Graefe and H. Kuno. Adaptive indexing for relational keys. In *ICDEW*, 2010.

[120] G. Graefe and H. Kuno. Self-selecting, self-tuning, incrementally optimized indexes. In *ICEDT*, 2010.

[121] C. Gray and D. Cheriton. *Leases: An efficient fault-tolerant mechanism for distributed file cache consistency*. 1989.

[122] J. N. Gray. An approach to decentralized computer systems. *IEEE Transactions on Software Engineering*, (6):684–692, 1986.

[123] H. Gupta, V. Harinarayan, A. Rajaraman, and J. D. Ullman. Index selection for olap. In *Data Engineering, 1997. Proceedings. 13th International Conference on*. IEEE, 1997.

[124] I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *JMLR*, 3, 2003.

[125] P. J. Haas and J. M. Hellerstein. Ripple joins for online aggregation. *ACM SIGMOD Record*, 28(2):287–298, 1999.

[126] P. J. Haas and C. König. A bi-level bernoulli scheme for database sampling. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 275–286. ACM, 2004.

[127] M. Hadjieleftheriou, X. Yu, N. Koudas, and D. Srivastava. Hashed samples: selectivity estimators for set similarity selection queries. *Proceedings of the VLDB Endowment*, 1(1):201–212, 2008.

[128] F. Halim and et. al. Stochastic database cracking: Towards robust adaptive indexing in main-memory column-stores. *PVLDB*, 5, 2012.

[129] J. Y. Halpern and J. Pearl. Causes and explanations: a structural-model approach. part i: causes. In *UAI*, 2001.

[130] J. Y. Halpern and J. Pearl. Causes and explanations: a structural-model approach. part ii: explanations. In *IJCAI*, 2001.

[131] F. M. Harper and J. A. Konstan. The movielens datasets: History and context. *Acm transactions on interactive intelligent systems (tiis)*, 5(4):19, 2016.

[132] A. B. Hastings. Distributed lock management in a transaction processing environment. In *Reliable Distributed Systems, 1990. Proceedings., Ninth Symposium on*, 1990.

[133] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *Acm Sigmod Record*, volume 26, pages 171–182. ACM, 1997.

[134] H. Herodotou and S. Babu. Xplus: a sql-tuning-aware query optimizer. *PVLDB*, 3, 2010.

[135] D. G. Horvitz and D. J. Thompson. A generalization of sampling without replacement from a finite universe. *Journal of the American statistical Association*, 47(260):663–685, 1952.

[136] K.-L. Hsiung, S.-J. Kim, and S. Boyd. Power control in lognormal fading wireless channels with uptime probability specifications via robust geometric programming. In *American Control Conference*, 2005.

[137] D. Huang, D. Y. Yoon, S. Pettie, and B. Mozafari. Join on sample: A theoretical guid for pracitioners. Technical report, 2019. https://web.eecs.umich.edu/~mozafari/php/data/uploads/approx-join-techreport.pdf.

[138] J. Huang, B. Mozafari, G. Schoenebeck, and T. Wenisch. A top-down approach to achieving performance predictability in database systems. In *SIGMOD*, 2017.

[139] J. Huang, B. Mozafari, and T. Wenisch. Statistical analysis of latency through semantic profiling. In *EuroSys*, 2017.

[140] R. Huang, R. Chirkova, and Y. Fathi. Two-stage stochastic view selection for data-analysis queries. In *Advances in Databases and Information Systems*, 2013.

[141] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX ATC*, 2010.

[142] S. Idreos, M. L. Kersten, and S. Manegold. Database cracking. In *CIDR*, 2007.

[143] S. Idreos, M. L. Kersten, and S. Manegold. Self-organizing tuple reconstruction in column-stores. In *SIGMOD*, 2009.

[144] E. Jahani, M. J. Cafarella, and C. Ré. Automatic optimization for mapreduce programs. *PVLDB*, 4, 2011.

[145] P. Jayanti, K. Tan, G. Friedland, and A. Katz. Bounding lamports bakery algorithm. In *International Conference on Current Trends in Theory and Practice of Computer Science*, 2001.

[146] C. Jermaine, S. Arumugam, A. Pol, and A. Dobra. Scalable approximate query processing with the dbo engine. *ACM Transactions on Database Systems (TODS)*, 33(4):23, 2008.

[147] C. Jermaine, A. Dobra, S. Arumugam, S. Joshi, and A. Pol. A disk-based join with probabilistic guarantees. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 563–574. ACM, 2005.

[148] H. Jula et al. Deadlock immunity: Enabling systems to defend against deadlocks. In *OSDI*, 2008.

[149] A. Kalia, M. Kaminsky, and D. G. Andersen. Using rdma efficiently for key-value services. In *ACM SIGCOMM Computer Communication Review*, 2014.

[150] A. Kalia, M. Kaminsky, and D. G. Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided (rdma) datagram rpcs. In *USENIX OSDI*, 2016.

[151] B. Kanagal, J. Li, and A. Deshpande. Sensitivity analysis and explanations for robust query evaluation in probabilistic databases. In *SIGMOD*, 2011.

[152] S. Kandula, A. Shanbhag, A. Vitorovic, M. Olma, R. Grandl, S. Chaudhuri, and B. Ding. Quickr: Lazily approximating complex adhoc queries in bigdata clusters. In *SIGMOD*, 2016.

[153] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat. Chronos: Predictable low latency for data center applications. In *Proceedings of the Third ACM Symposium on Cloud Computing*, 2012.

[154] E. Keogh, J. Lin, A. W. Fu, and H. Van Herle. Finding unusual medical time-series subsequences: Algorithms and applications. *Information Technology in Biomedicine, IEEE Transactions on*, 10(3):429–439, 2006.

[155] N. Khoussainova, M. Balazinska, and D. Suciu. PerfXplain: Debugging mapreduce job performance. *PVLDB*, 2012.

[156] A. Kim, E. Blais, A. G. Parameswaran, P. Indyk, S. Madden, and R. Rubinfeld. Rapid sampling for visualizations with ordering guarantees. *PVLDB*, 2015.

[157] A. Kipf, T. Kipf, B. Radke, V. Leis, P. Boncz, and A. Kemper. Learned cardinalities: Estimating correlated joins with deep learning. *arXiv preprint arXiv:1809.00677*, 2018.

[158] A. C. Konig and S. U. Nabar. Scalable exploration of physical database design. In *ICDE*, 2006.

[159] M. Kormilitsin, R. Chirkova, Y. Fathi, and M. Stallmann. View and index selection for query-performance improvement: quality-centered algorithms and heuristics. In *CIKM*, 2008.

[160] N. P. Kronenberg, H. M. Levy, and W. D. Strecker. Vaxcluster: a closely-coupled distributed system. 1986.

[161] B.-J. Kwak, N.-O. Song, and L. E. Miller. Performance analysis of exponential backoff. 2005.

[162] E. Kwan, S. Lightstone, A. Storm, and L. Wu. Automatic configuration for ibm db2 universal database. In *Proc. of IBM Perf Technical Report*, 2002.

[163] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear. The vertica analytic database: C-store 7 years later. *PVLDB*, 5(12), 2012.

[164] L. Lamport. A new solution of dijkstra's concurrent programming problem. *Communications of the ACM*, 1974.

[165] L. Lamport et al. Paxos made simple. *ACM Sigact News*, 2001.

[166] K.-H. Lee and G.-J. Park. A global robust optimization using kriging based approximation model. *JSME*, 49, 2006.

[167] V. Leis, B. Radke, A. Gubichev, A. Kemper, and T. Neumann. Cardinality estimation done right: Index-based join sampling. In *CIDR*, 2017.

[168] F. Li, S. Das, M. Syamala, and V. R. Narasayya. Accelerating relational databases by leveraging remote memory and rdma. In *SIGMOD*, 2016.

[169] F. Li, B. Wu, K. Yi, and Z. Zhao. Wander join: Online aggregation via random walks. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 615–629, 2016.

[170] F. Li, B. Wu, K. Yi, and Z. Zhao. Wander join and XDB: online aggregation via random walks. *ACM Trans. Database Syst.*, 44(1):2:1–2:41, 2019.

[171] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing*, 2014.

[172] J. Li, T. D. Le, L. Liu, J. Liu, Z. Jin, and B. Sun. Mining causal association rules. In *Data Mining Workshops (ICDMW), 2013 IEEE 13th International Conference on*, pages 114–123. IEEE, 2013.

[173] J. Li, N. K. Sharma, D. R. Ports, and S. D. Gribble. Tales of the tail: Hardware, os, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing*, 2014.

[174] W. Liang, H. Wang, and M. E. Orlowska. Materialized view selection under the maintenance time constraint. *Data & Knowledge Engineering*, 37(2), 2001.

[175] R. G. Lorenz and S. P. Boyd. Robust minimum variance beamforming. *Signal Processing, IEEE Transactions on*, 53(5), 2005.

[176] G. Luo, C. J. Ellmann, P. J. Haas, and J. F. Naughton. A scalable hash ripple join algorithm. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 252–262. ACM, 2002.

[177] G. Luo, J. F. Naughton, C. J. Ellmann, and M. W. Watzke. Transaction reordering. *Data & Knowledge Engineering*, 2010.

[178] A. A. Mahimkar, Z. Ge, A. Shaikh, J. Wang, J. Yates, Y. Zhang, and Q. Zhao. Towards automated performance diagnosis in a large IPTV network. In *ACM SIGCOMM Computer Communication Review*, volume 39, pages 231–242. ACM, 2009.

[179] C. Maier and et. al. Parinda: an interactive physical designer for postgresql. In *ICEDT*, 2010.

[180] I. Mami and Z. Bellahsene. A survey of view selection methods. *SIGMOD Record*, 41(1), 2012.

[181] V. Markl and et. al. Robust query processing through progressive optimization. In *SIGMOD*, 2004.

[182] A. Meliou, W. Gatterbauer, and D. Suciu. Bringing provenance to its full potential using causal reasoning. In *TaPP*, 2011.

[183] C. Mitchell, Y. Geng, and J. Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *USENIX ATC*, 2013.

[184] K. Mori. Autonomous decentralized systems: Concept, data field architecture and future trends. In *Proceedings ISAD 93: International Symposium on Autonomous Decentralized Systems*, pages 28–34. IEEE, 1993.

[185] B. Mozafari, C. Curino, A. Jindal, and S. Madden. Performance and resource modeling in highly-concurrent OLTP workloads. In *SIGMOD*, 2013.

[186] B. Mozafari, C. Curino, and S. Madden. DBSeer: Resource and performance prediction for building a next generation database cloud. In *CIDR*, 2013.

[187] B. Mozafari, E. Z. Y. Goh, and D. Y. Yoon. Cliffguard: A principled framework for finding robust database designs. In *SIGMOD*, 2015.

[188] B. Mozafari, E. Z. Y. Goh, and D. Y. Yoon. CliffGuard: A principled framework for finding robust database designs. In *SIGMOD*, 2015.

[189] B. Mozafari, E. Z. Y. Goh, and D. Y. Yoon. Cliffguard: An extended report. Technical report, University of Michigan, Ann Arbor, 2015.

[190] B. Mozafari and N. Niu. A handbook for building an approximate query engine. *IEEE Data Engineering Bulletin*, 2015.

[191] B. Mozafari, J. Ramnarayan, S. Menon, Y. Mahajan, S. Chakraborty, H. Bhanawat, and K. Bachhav. SnappyData: A unified cluster for streaming, transactions, and interactive analytics. In *CIDR*, 2017.

[192] S. Narravula, A. Marnidala, A. Vishnu, K. Vaidyanathan, and D. K. Panda. High performance distributed lock management services using network-based remote atomic operations. In *Seventh IEEE International Symposium on Cluster Computing and the Grid*, 2007.

[193] J. Nelson et al. Latency-tolerant software distributed shared memory. In *USENIX ATC*, 2015.

[194] R. Pagh, M. Stöckel, and D. P. Woodruff. Is min-wise hashing optimal for summarizing set intersection? In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS'14, Snowbird, UT, USA, June 22-27, 2014*, pages 109–120, 2014.

[195] G. Palermo, C. Silvano, and V. Zaccaria. Robust optimization of soc architectures: A multi-scenario approach. In *Embedded Systems for Real-Time Multimedia*, 2008.

[196] N. Pansare, V. R. Borkar, C. Jermaine, and T. Condie. Online aggregation for large mapreduce jobs. *Proc. VLDB Endow*, 4(11):1135–1145, 2011.

[197] S. Papadomanolakis and A. Ailamaki. An integer linear programming approach to database design. In *ICDE Workshop*, 2007.

[198] Y. Park, B. Mozafari, J. Sorenson, and J. Wang. Verdictdb: universalizing approximate query processing. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1461–1476. ACM, 2018.

[199] D. Patil, S. Yun, S.-J. Kim, A. Cheung, M. Horowitz, and S. Boyd. A new method for design of robust digital circuits. In *ISQED*, 2005.

[200] J. Pearl, T. Verma, et al. *A theory of inferred causation*. Morgan Kaufmann San Mateo, CA, 1991.

[201] J.-P. Pellet and A. Elisseeff. Using markov blankets for causal structure learning. *The Journal of Machine Learning Research*, 9:1295–1342, 2008.

[202] T. Pitoura and P. Triantafillou. Self-join size estimation in large-scale distributed data systems. In *2008 IEEE 24th International Conference on Data Engineering*, pages 764–773. IEEE, 2008.

[203] C. Qin and F. Rusu. Pf-ola: a high-performance framework for parallel online aggregation. *Distributed and Parallel Databases*, pages 1–39, 2013.

[204] S. Rahman, M. Aliakbarpour, H. Kong, E. Blais, K. Karahalios, A. G. Parameswaran, and R. Rubinfeld. I've seen "enough": Incrementally improving visualizations to support rapid decision making. *PVLDB*, 2017.

[205] R. Rajwar and J. R. Goodman. Transactional lock-free execution of lock-based programs. In *ACM SIGOPS Operating Systems Review*, 2002.

[206] V. Raman and et. al. Constant-time query processing. In *ICDE*, 2008.

[207] J. Ramnarayan, B. Mozafari, S. Menon, S. Wale, N. Kumar, H. Bhanawat, S. Chakraborty, Y. Mahajan, R. Mishra, and K. Bachhav. SnappyData: A hybrid transactional analytical store built on spark. In *SIGMOD*, 2016.

[208] K. Rashmi, M. Chowdhury, J. Kosaian, I. Stoica, and K. Ramchandran. Ec-cache: Load-balanced, low-latency cluster caching with online erasure coding. In *USENIX OSDI*, 2016.

[209] R. Recio, P. Culley, D. Garcia, J. Hilland, and B. Metzler. An rdma protocol specification. Technical report, IETF Internet-draft draft-ietf-rddp-rdmap-03. txt (work in progress), 2005.

[210] K. Ren, A. Thomson, and D. J. Abadi. Vll: a lock manager redesign for main memory database systems. *The VLDB Journal*, 2015.

[211] W. Rödiger, T. Mühlbauer, A. Kemper, and T. Neumann. High-speed query processing over high-speed networks. 2015.

[212] A. Rogge-Solti and G. Kasneci. Temporal anomaly detection in business processes. In *Business Process Management*. 2014.

[213] S. Roy, A. C. König, I. Dvorkin, and M. Kumar. PerfAugur: Robust diagnostics for performance anomalies in cloud services. In *ICDE*, 2015.

[214] S. Sarawagi. Explaining differences in multidimensional aggregates. In *VLDB*, 1999.

[215] K. Schnaitter, S. Abiteboul, T. Milo, and N. Polyzotis. On-line index selection for shifting workloads. In *ICDE Workshop*, 2007.

[216] D. Shin et al. Dynamic interval polling and pipelined post i/o processing for low-latency storage class memory. In *HotStorage*, 2013.

[217] A. Shukla, P. Deshpande, J. F. Naughton, et al. Materialized view selection for multidimensional datasets. In *VLDB*, volume 98, 1998.

[218] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *MSST*, 2010.

[219] T. Siddiqui, A. Kim, J. Lee, K. Karahalios, and A. G. Parameswaran. Effortless data exploration with zenvisage: An expressive and interactive visual analytics system. *PVLDB*, 2016.

[220] C. Silverstein, S. Brin, R. Motwani, and J. Ullman. Scalable techniques for mining causal structures. *Data Mining and Knowledge Discovery*, 4(2-3):163–192, 2000.

[221] SQL Server 2014. *Database Engine Tuning Advisor*, 2014.

[222] M. Stonebraker and A. Weisberg. The voltdb main memory dbms. *IEEE Data Eng. Bull.*, 2013.

[223] H. Su, M. Zait, V. Barrière, J. Torres, and A. Menck. Approximate aggregates in oracle 12c, 2016.

[224] A. Swami and K. B. Schiefer. On the estimation of join result sizes. In *International Conference on Extending Database Technology*, pages 287–300. Springer, 1994.

[225] Z. A. Talebi, R. Chirkova, and Y. Fathi. An integer programming approach for the view and index selection problem. *Data & Knowledge Engineering*, 83, 2013.

[226] Y. Tan, X. Gu, and H. Wang. Adaptive system anomaly prediction for large-scale hosting infrastructures. In *PODC*, 2010.

[227] G. Taubenfeld. The black-white bakery algorithm and related bounded-space, adaptive, local-spinning and fifo algorithms. *Distributed Computing*, 2004.

[228] B. Tian, J. Huang, B. Mozafari, G. Schoenebeck, and T. Wenisch. Contention-aware lock scheduling for transactional databases. *PVLDB*, 2018.

[229] Q. T. Tran, I. Jimenez, R. Wang, N. Polyzotis, and A. Ailamaki. Rita: An index-tuning advisor for replicated databases. *arXiv preprint arXiv:1304.1411*, 2013.

[230] J. Tu, K. K. Choi, and Y. H. Park. A new study on reliability-based design optimization. *Journal of Mechanical Design*, 121(4), 1999.

[231] G. Valentin and et. al. Db2 advisor: An optimizer smart enough to recommend its own indexes. In *ICDE*, 2000.

[232] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1009–1024. ACM, 2017.

[233] A. van der Vaart. *Asymptotic statistics*, volume 3. Cambridge university press, 2000.

[234] R. Varadarajan, V. Bharathan, A. Cary, J. Dave, and S. Bodagala. Dbdesigner: A customizable physical design tool for vertica analytic database. In *ICDE*, 2014.

[235] D. Vengerov, A. C. Menck, M. Zaït, and S. Chakkappen. Join size estimation subject to filter conditions. *PVLDB*, 8(12):1530–1541, 2015.

[236] A. Vulimiri, O. Michel, P. Godfrey, and S. Shenker. More is less: reducing latency via redundancy. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, 2012.

[237] C. A. Waldspurger. Lottery and stride scheduling: Flexible proportional-share resource management. 1995.

[238] L. Wei, N. Kumar, V. N. Lolla, E. J. Keogh, S. Lonardi, and C. A. Ratanamahatana. Assumption-free anomaly detection in time series. In *SSDBM*, volume 5, pages 237–242, 2005.

[239] L. Wei, W. Qian, A. Zhou, W. Jin, and X. Jeffrey. Hot: Hypergraph-based outlier test for categorical data. In *AKDDM*. 2003.

[240] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast in-memory transaction processing using rdma and htm. In *SOSP*, 2015.

[241] G. Weikum, C. Hasse, A. Mönkeberg, and P. Zabback. The comfort automatic tuning project. *Information systems*, 19(5):381–432, 1994.

[242] E. Wu and S. Madden. Scorpion: Explaining away outliers in aggregate queries. *PVLDB*, 6, 2013.

[243] S. Wu, B. C. Ooi, and K.-L. Tan. Continuous sampling for online aggregation over multiple queries. In *SIGMOD*, 2010.

[244] W. Wu, J. F. Naughton, and H. Singh. Sampling-based query re-optimization. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1721–1736. ACM, 2016.

[245] C. Yan and A. Cheung. Leveraging lock contention to improve oltp application performance. 2016.

[246] D. Y. Yoon, M. Chowdhury, and B. Mozafari. Distributed lock management with rdma: Decentralization without starvation. In *SIGMOD*, 2018.

[247] D. Y. Yoon, B. Mozafari, and D. P. Brown. DBSeer: Pain-free database administration through workload intelligence. *PVLDB*, 2015.

[248] D. Y. Yoon, N. Niu, and B. Mozafari. DBSherlock: A performance diagnostic tool for transactional databases. In *SIGMOD*, 2016.

[249] J. X. Yu, W. Qian, H. Lu, and A. Zhou. Finding centric local outliers in categorical/numerical spaces. *KAIS*, 9, 2006.

[250] J. X. Yu, X. Yao, C.-H. Choi, and G. Gou. Materialized view selection as constrained evolutionary optimization. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 33(4), 2003.

[251] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. *HotCloud*, 2010.

[252] K. Zeng, S. Gao, J. Gu, B. Mozafari, and C. Zaniolo. ABS: a system for scalable approximate queries with accuracy guarantees. In *SIGMOD*, 2014.

[253] K. Zeng, S. Gao, B. Mozafari, and C. Zaniolo. The analytical bootstrap: a new method for fast error estimation in approximate query processing. In *SIGMOD*, 2014.

[254] L. Zhang, Y. Chen, Y. Dong, and C. Liu. Lock-visor: An efficient transitory co-scheduling for mp guest. In *ICPP*, 2012.

[255] Y. Zhang. General robust-optimization formulation for nonlinear programming. *JOTA*, 132, 2007.

[256] Z. Zhao, R. Christensen, F. Li, X. Hu, and K. Yi. Random sampling over joins revisited. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1525–1539. ACM, 2018.

[257] Z. Zhao, R. Christensen, F. Li, X. Hu, and K. Yi. Random sampling over joins revisited. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1525–1539, 2018.

[258] Z. Zhao, E. Zgraggen, L. D. Stefani, C. Binnig, E. Upfal, and T. Kraska. Safe visual data exploration. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, 2017.

[259] D. C. Zilio and et. al. Recommending materialized views and indexes with the ibm db2 design advisor. In *ICAC*, 2004.