Elastic Database Systems

by

Rebecca Taft

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2017

Author . . . . . .
Department of Electrical Engineering and Computer Science
August 31, 2017

Certified by . . . . . .
Michael R. Stonebraker
Adjunct Professor of Computer Science
Thesis Supervisor

Accepted by . . . . . .
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Elastic Database Systems

by

Rebecca Taft

Submitted to the Department of Electrical Engineering and Computer Science
on August 31, 2017, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Electrical Engineering and Computer Science

Abstract

Distributed on-line transaction processing (OLTP) database management systems (DBMSs) are a critical part of the operation of large enterprises. These systems often serve time-varying workloads due to daily, weekly or seasonal fluctuations in load, or because of rapid growth in demand due to a company's business success. In addition, many OLTP workloads are heavily skewed to "hot" tuples or ranges of tuples. For example, the majority of NYSE volume involves only 40 stocks. To manage such fluctuations, many companies currently provision database servers for peak demand. This approach is wasteful and not resilient to extreme skew or large workload spikes. To be both efficient and resilient, a distributed OLTP DBMS must be elastic; that is, it must be able to expand and contract its cluster of servers as demand fluctuates, and dynamically balance load as hot tuples vary over time.

This thesis presents two elastic OLTP DBMSs, called E-Store and P-Store, which demonstrate the benefits of elasticity for distributed OLTP DBMSs on different types of workloads. E-Store automatically scales the database cluster in response to demand spikes, periodic events, and gradual changes in an application's workload, but it is particularly well-suited for managing hot spots. In contrast to traditional single-tier hash and range partitioning strategies, E-Store manages hot spots through a two-tier data placement strategy: cold data is distributed in large chunks, while smaller ranges of hot tuples are assigned explicitly to individual nodes. P-Store is an elastic OLTP DBMS that is designed for a subset of OLTP applications in which load varies predictably. For these applications, P-Store performs better than reactive systems like E-Store, because P-Store uses predictive modeling to reconfigure the system in advance of predicted load changes.

The experimental evaluation shows the efficacy of the two systems under variations in load across a cluster of machines. Compared to single-tier approaches, E-Store improves throughput by up to 130% while reducing latency by 80%. On a predictable workload, P-Store outperforms a purely reactive system by causing 72% fewer latency violations, and achieves performance comparable to static allocation for peak demand while using 50% fewer servers.

Thesis Supervisor: Michael R. Stonebraker
Title: Adjunct Professor of Computer Science

Acknowledgments

First and foremost, I would like to thank my PhD Advisor, Michael Stonebraker. Over the last five years, Mike has taught me how to choose interesting and relevant research problems, write conference papers that stand up to scrutiny, and communicate effectively. Mike's ability to see "the view at 10,000 feet" of complex problems allowed me to leave almost all of our meetings with a sense of purpose and marching orders to "make it happen", even if I had felt hopelessly lost going in.

I would also like to thank the other two members of my thesis committee, Sam Madden and Frans Kaashoek. Sam and Frans have both been important mentors and advisors during my time at MIT, and provided valuable feedback and ideas to improve this thesis. Sam's comments helped me solidify the motivation for the work, and Frans' comments helped me to clarify the limitations and explain some of the surprising results.

Although not a member of my thesis committee, David DeWitt has been an essential advisor and collaborator throughout much of my time at MIT. I am especially grateful for our many long discussions about my career options after MIT.

Although this thesis has only one author, the work was a group effort and I could not have done it without my collaborators. (In alphabetical order) Ashraf Aboulnaga, Francisco Andrade, Aaron Elmore, Nosayba El-Sayed, Yu Lu, Essam Mansour, Ricardo Mayerhofer, Andy Pavlo, Jennie Rogers, Marco Serafini, and Mike Stonebraker are my coauthors on the E-Store (VLDB 2015) and P-Store (under submission) conference papers, and they deserve much of the credit for this research.

My parents, Phyllis and Tucker, have been an incredible source of support throughout my PhD, especially during the last three years when I lived with them. Dinner table discussions with my dad often delved into research problems, and gave me several important insights. My partner, Raul, has also been a pillar of support, and gave me excellent constructive criticism on early drafts of this thesis.

There are many other people who have contributed to my wellbeing and success at MIT. The entire MIT Database Group both past and present has been instrumental in my growth as a researcher and public speaker. The MIT Rowing Club and Cycling Team have both

been an important source of support and balance outside of academics. And I am grateful to the many other friends and family members who have stood by me and kept me sane throughout the ups and downs of PhD life.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Due to the proliferation of public cloud offerings like Amazon AWS, Microsoft Azure and Google Cloud, today's web developers can build applications that easily scale to millions of concurrent users with the click of a button. For example, startups such as Airbnb, Duolingo, and Lyft have all used Amazon's AWS offerings to manage their rapid growth [3]. Accordingly, internet users have come to expect rich functionality with lightning-fast response times, even when millions of other users are trying to access the same content.

A key piece of technology enabling this scalability and fueling the productivity of web developers is the modern, distributed database management system (DBMS). DBMSs generally constitute the "back-end" of these web applications, and are responsible for storing the data associated with the users and content of the site. For example, the items in your online retail shopping cart, your bank account balance, and your airline reservations are all likely stored in a DBMS. These examples are representative of a class of applications commonly known as *on-line transaction processing (OLTP)* applications, because they largely interact with the DBMS through many real-time, small transactions [42]. Transactions are operations on the DBMS that read and/or modify the data, such as adding an item to a shopping cart, viewing a bank account balance, or making a flight reservation. The value of performing these operations as transactions in an OLTP DBMS is that the database provides certain correctness guarantees, freeing the developer to focus on other things like building new features. These guarantees are known by the acronym ACID: Atomicity, Consistency, Isolation, and Durability [79, 42].

Atomicity: This refers to the all-or-nothing nature of transactions. For example, if Alice sends a wire transfer of $100 to Bob, the transaction must (1) deduct $100 from Alice's bank account and (2) add $100 to Bob's account. Both actions must succeed, or else the entire transfer should fail.

Consistency: This guarantee ensures that data is never corrupted, and any declared constraints always hold. For example, if a column in a table is declared to be unique (e.g., an employee ID), it will not be possible to insert two records with the same value for that column.

Isolation: This means that two concurrently running transactions will not interfere with each other, even if they access the same data. It must appear as if the two transactions completed serially even if their operations were interleaved. This is one of the reasons that read-only transactions are also considered transactions; read-only transactions must not see data from a concurrently-running update transaction that has been only partially completed.

Durability: This ensures that once data has been inserted in a database and successfully committed, no crashes, hardware failures or other disasters will cause it to be erased unintentionally.

The ACID properties are essential for many applications, and developers have traditionally relied on the DBMS to provide these guarantees. However, many traditional SQL-based OLTP DBMSs do not easily scale to the required levels needed to serve modern web applications, and therefore cannot meet the throughput and latency requirements of these applications. Maximizing throughput (rate of transaction execution) while keeping latency (delay per transaction) below a given threshold is critical to the success of most OLTP applications, since these performance metrics are directly related to the volume of users that can be served and the response time of the application. As a result, NoSQL DBMSs such as MongoDB [18], Cassandra [57] and Amazon's DynamoDB [25] have gained in popularity due to their impressive scalability and simplicity. But NoSQL systems do not provide all of the ACID guarantees, leaving application developers to build this functionality into applications themselves. Building ACID functionality on-the-fly is a difficult and error-prone process, making NoSQL systems often more trouble than they are worth for such applications.

For applications not willing to compromise on consistency guarantees, there are several other techniques that researchers and companies have begun to use to achieve scalability [9, 101, 74]. For example, due to the large amount of RAM available in modern servers, it is now possible to store most of the data in an OLTP database in main memory, allowing modern DBMSs to remove much of the overhead of older disk-based systems [44]. Logical logging, single-threaded execution, and latch-free data structures are other techniques that have improved the performance of several modern OLTP DBMSs [48, 26, 97].

In order to achieve even higher levels of scalability, many new DBMSs are making use of older ideas that have new relevance in the age of cloud computing. For example, many new systems employ a *shared nothing* architecture [88], in which data is distributed across a cluster of machines (also called servers or nodes) which do not share either memory or disk. Often the data is *partitioned* (also known as *sharded*), meaning subsets of the tuples (records) in the database are assigned to specific servers, and no single server contains all the data. Because servers do not share data, if a transaction requires access to a specific set of tuples (e.g., to read or update the data), it must be executed on the server(s) containing those tuples. To make it easy to route transactions to the correct server, a hash function is typically used to map tuples to servers. It is also possible to perform this mapping at a finer granularity, e.g. with a lookup table mapping individual tuples or key ranges to servers [93]. The advantage of the shared nothing partitioned model is that if a transaction only needs to access data on a single server, no communication is required with any other servers. This allows many transactions to be executed in parallel if they touch different data, and is the key feature enabling scalability. For workloads in which transactions are uniformly distributed across the database and each transaction touches a small amount of data, shared nothing partitioned systems can scale almost linearly (i.e., capacity increases linearly with cluster size).

One downside of many modern distributed DBMSs is that they are difficult to use in practice, because changing the configuration of the database to scale out and add servers is often a manual process. Furthermore, many OLTP applications are subject to workloads that vary considerably over time, requiring frequent reconfiguration of the database. Some NoSQL systems such as Amazon's DynamoDB allow "auto-scaling", but DynamoDB only

works for applications that do not require the flexibility and guarantees of SQL, and is "ideally suited for request patterns that are uniform, predictable, with sustained high and low throughput usage that lasts for several minutes to hours" [4]. Clearly, DynamoDB will not be a good solution for OLTP applications that are subject to highly variable and spiky traffic patterns. This extreme variability is especially prevalent in web-based services, which handle large numbers of requests whose volume may depend on factors such as the weather or social media trends. For example, an e-commerce site might become overwhelmed during a holiday sale. Moreover, specific items within the database can suddenly become popular, such as when a review of a book on a TV show generates a deluge of orders in on-line bookstores. This phenomenon in which a small number of items receive a disproportionately large number of the transactions is known as *skew*. As such, it is important that a DBMS be resilient to both load spikes and skew.

This thesis focuses on making DBMSs resilient to such variability using several techniques collectively known as *database elasticity*. An ideal elastic database adapts to changes in an application's workload **without manual intervention** to ensure that application throughput and latency requirements are met, **while continuing to preserve transactional ACID guarantees.** It is this last part that makes this problem particularly challenging. NoSQL systems are able to scale a DBMS cluster easily because they do not support full SQL transactions. Eliminating manual intervention is also essential to ensure that the system can react immediately to a change in the workload; if the DBMS has to wait for a human to perform corrective actions, the event that caused the problem may have passed.

The techniques discussed in this thesis enable an OLTP DBMS to monitor its workload, and automatically add (or remove) servers as soon as it determines that additional capacity is needed (or no longer needed) to meet throughput and latency requirements. If a workload follows a predictable pattern, the system can detect this pattern and scale out proactively to achieve even better performance. Besides adding or removing servers to adapt to changes in the aggregate workload, the elasticity techniques discussed balance the workload across servers to adapt to fine-grained changes in access patterns. This load balancing is achieved by changing the way data is partitioned across servers (i.e. changing the partitioning hash function and moving data accordingly) in order to ensure that every server in the database

can be highly utilized for transaction processing.

At first glance, some of these techniques may seem unnecessary to all but the largest companies, since many of today's OLTP applications can be served by highly memory-optimized single-node DBMSs [26, 97]. But as more and more companies move to the cloud and make use of Software-as-a-Service (SaaS) offerings (e.g., Salesforce [104]), distributed DBMSs will be essential to a larger fraction of applications. Furthermore, as the global internet traffic continues to rise [32] and transactions are increasingly generated by sensors and algorithms [10], the need for databases that can handle extremely high throughput will only increase. Database elasticity will be an important tool to manage this growth and ensure that distributed databases are as efficient as possible.

The remainder of this chapter examines the industry-standard approach for managing workload variability, and describes in detail how database elasticity is a better solution. Next it introduces the research contributions, and presents the two systems built as part of this thesis. These systems implement several novel elasticity techniques and demonstrate their effectiveness on real and synthetic workloads. The chapter concludes by summarizing the contributions of the research and outlining the rest of the thesis.

1.1 The Status Quo

To date, the way that administrators deal with fluctuations in demand on an OLTP DBMS is primarily a manual process. Too often it is a struggle to increase capacity and remove system bottlenecks faster than the DBMS load increases [33]. This is especially true for applications that require strong transaction guarantees without service interruptions. To manage the risk of unanticipated load fluctuations, companies frequently provision computing resources for some multiple of their routine load, since the peak demand may range from 2–10$\times$ the average [7]. This leaves resources underutilized for a substantial fraction of the time.

Given this pervasive underutilization, there is a desire in many enterprises to consolidate OLTP applications onto a smaller collection of powerful servers, whether using a public cloud platform or an internal cloud. This multi-tenancy promises to decrease

over-provisioning for OLTP applications and introduce economies of scale such as shared personnel (e.g., system administrators). But unless the demand for these co-located applications is uncorrelated, the net effect of multi-tenancy might still allow severe fluctuations in load.

A more robust solution is to enable DBMSs to adapt to workload variation by adding resources dynamically. But most companies do not dynamically provision computing resources for database systems, even though dynamic provisioning is often used for stateless web services [49]. OLTP databases are difficult to reconfigure because it is necessary to copy data between servers in a transactionally-consistent manner while keeping the database live and able to accept new requests. Insert-heavy workloads such as Internet-of-Things (IoT) and streaming applications may be able to send new data to new servers without moving existing data, but most other workloads will require some amount of data to be moved in order to handle increased accesses to existing data. Depending on the amount of data that must be migrated, reconfiguration can take anywhere from a few seconds to ten minutes or more [28]. During this reconfiguration period, the system may experience degraded performance in the form of higher transaction latency or a higher transaction abort rate.

Despite the challenges inherent in reconfiguring OLTP DBMSs, companies are starting to pursue dynamic provisioning because the status quo is no longer acceptable. As a case study, let us examine the Brazilian company B2W Digital (B2W) [11], which has been a key collaborator in this research. B2W is the largest online retailer in South America, sometimes referred to as "The Amazon of South America". They own four major brands including Americanas.com, Shoptime.com, Submarino.com and SouBarato.com. Americanas is the largest of the four websites and, similar to Amazon, sells everything from books to clothes to household appliances. B2W is motivated to participate in this research because they have a heavy and extremely variable OLTP workload. Load on their databases at peak times can be more than 10x the load at other times. On Black Friday, the day after Thanksgiving in the United States, the load is so much higher than normal that B2W actually reconfigures the database clusters to use more powerful servers several days before the event. This scale-up process requires significant manual effort and weeks to months of

planning. B2W realizes that there are many other opportunities to save money by limiting computing resources throughout the year, but the manual effort is too great. A system that automatically reconfigures itself without human intervention has the potential to save them a significant amount of money, especially as their company grows and requires ever-larger database clusters.

Researchers have suggested some ideas for automatic OLTP database repartitioning for load balancing purposes [82, 95, 31], but most of these solutions do not handle cases with extreme skew, and many do not support elasticity at all. Google's Spanner is at the cutting edge of industrial solutions since it enables automatic load balancing by "resharding" with fine-grained partitioning key ranges, but Google's recent conference paper does not provide details of the load balancing algorithm or monitoring infrastructure [12]. In the public cloud offering of Spanner, the decision to add or remove nodes is manual; there is no auto-scale option [40].

1.2 Database Elasticity

Pervasive overprovisioning is a waste of resources, and might not be sufficient to ensure good performance if load exceeds capacity due to an excessively large spike. Furthermore, overprovisioning does little to prevent performance issues caused by hot spots. Database elasticity is challenging but necessary because OLTP applications can incur several types of workload variability and skew that each require different solutions. Examples of these include:

Hot Spots: In many OLTP applications, the rate that transactions access certain individual tuples or small key ranges within a table is often "skewed". For example, 40–60% of the volume on the New York Stock Exchange (NYSE) occurs on just 40 out of ∼4000 stocks [72]. This phenomenon also appears in online retail stores, where some items are much more popular than others. For example, an estimated 55 million copies of the 70 books selected for Oprah's Book Club were sold due to the "Oprah Effect" [69].

Time-Varying Skew: Multi-national customer support applications tend to exhibit a "follow the sun" cyclical workload. Here, workload demand shifts around the globe fol-

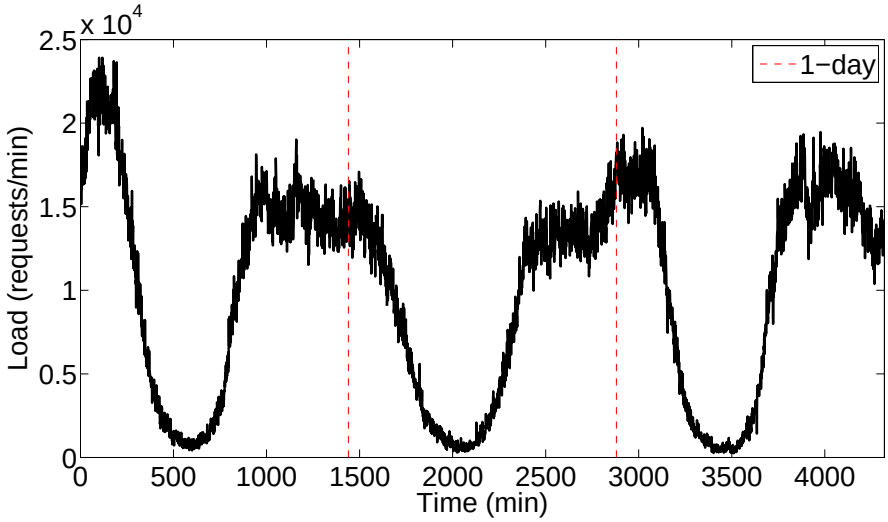


Figure 1-1: Load on one of B2W's databases over three days in terms of requests per minute. Load peaks during daytime hours and dips at night.

lowing daylight hours when most people are awake. This means that the load in any geographic area will resemble a sine wave over the course of a day. For example, Figure 1-1 shows the database workload over three days of B2W Digital, the Brazilian company introduced in Section 1.1. As can be seen, the peak load is about 10× the trough. If companies like B2W could take advantage of database elasticity to use exactly as many computing resources as needed to manage their workload, they could reduce the average number of servers needed for their database by about half. In the case of a private cloud, these servers could be temporarily repurposed for some other application in the organization. In a public cloud, the reduction in servers translates directly into reduced expenses for the organization. Time-dependent workloads may also have cyclic skew with other periodicities. For example, an on-line application to reserve camping sites will have seasonal variations in load, with summer months being much busier than winter months.

Load Spikes: A DBMS may incur short periods when the number of requests increases significantly over the normal expected volume. For example, the volume on the NYSE during the first and last ten minutes of the trading day is an order of magnitude higher than at other times. Such surges may be predictable, as in the NYSE system, or the product of "one-shot" effects. One encyclopedia vendor experienced this problem when it put its content on the web and the initial flood of users after the announcement caused a huge load

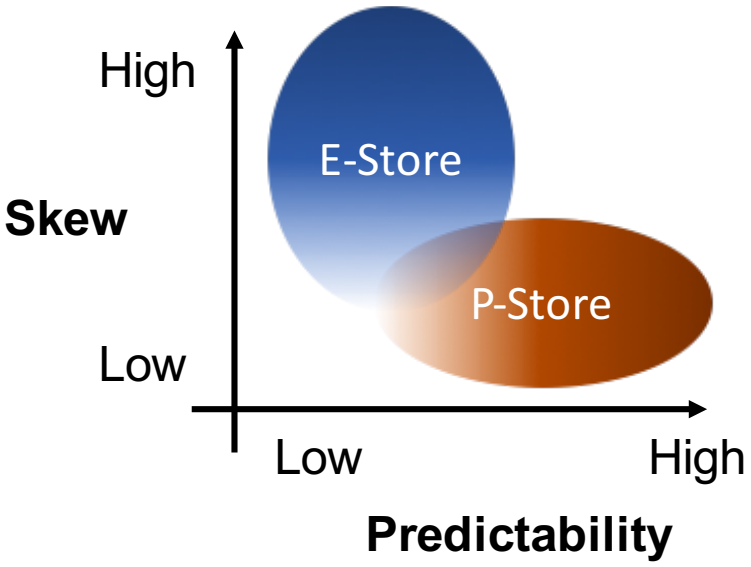


Figure 1-2: The best conditions for the E-Store and P-Store systems as a function of workload skew and predictability

spike that crashed the service [36].

The Hockey Stick Effect: Web-based startups often see a "hockey stick" of traffic growth. When (and if) their application becomes popular, they will have an exponential increase in traffic that leads to crushing demand on its DBMS. This pitfall also impacts established companies when they roll out a new product.

Given these issues, it is essential that an OLTP DBMS be *elastic*. That is, it must automatically adapt to workload changes without manual intervention while preserving ACID guarantees. For an OLTP DBMS with a shared nothing partitioned architecture, this involves adding or removing servers to increase or decrease capacity, and rebalancing data across nodes. If hot spots are causing one server to be overloaded, the hot tuples must be split up and redistributed across the cluster. For unexpected changes in the workload such as load spikes or the hockey stick effect, the DBMS must quickly react to reconfigure so it can continue to meet throughput and latency requirements. For predictable changes such as time varying skew, the DBMS should reconfigure proactively so that reconfiguration will complete in advance of predicted load increases.

This thesis presents two different systems for elastically scaling an OLTP DBMS, depicted in Figure 1-2. The first system, called E-Store, is the most general. It can adapt to

all of the different types of workload variation and skew described above. E-Store is ideal for managing high skew and reacting quickly to unexpected load spikes. But some OLTP applications have lower levels of skew and exhibit predictable, "follow the sun" behavior. For these applications, the second system, called P-Store, is a better fit. P-Store uses predictive modeling to proactively reconfigure the DBMS, and can achieve superior performance to E-Store for predictable workloads. Figure 1-2 summarizes the best conditions for each system as a function of workload skew and predictability. Although these systems do not currently support predictable workloads with high skew, the ideas of E-Store and P-Store are complementary, and future work should combine these systems to support a wider variety of workloads. The next two sections introduce each system in more detail.

1.3 Fine-Grained Partitioning for Reactive Elasticity

The first system presented in this thesis is called E-Store, a planning and reconfiguration system for shared-nothing, distributed DBMSs optimized for transactional workloads. The main contribution of E-Store is a comprehensive framework that addresses many of the types of workload variation and skew discussed in Section 1.2. E-Store is particularly well suited for cases of high skew because it can detect individual tuples that are frequently accessed and assign adequate resources for their associated transactions by placing them explicitly on servers with sufficient capacity. Instead of monitoring and migrating data at the granularity of pre-defined large chunks as some existing systems do [68, 82], E-Store dynamically alters chunks by extracting their *hot tuples*, which are considered as separate "singleton" chunks. Introducing this *two-tiered* approach combining fine-grained hot chunks and coarse-grained cold chunks is the main technical contribution enabling E-Store to reach its goals. E-Store supports automatic on-line hardware reprovisioning that enables a DBMS to move its tuples between existing nodes to break up hotspots, as well as to scale the size of the DBMS's cluster.

E-Store identifies skew using a suite of monitoring tools. First it identifies when load surpasses a threshold using a lightweight algorithm. When this occurs, a second monitoring component is triggered that is integrated in the DBMS to track tuple-level access patterns.

This information is used in E-Store's novel two-tier data placement scheme that assigns tuples to nodes based on their access frequency. This approach first distributes hot tuples one at a time throughout the cluster. Then it allocates cold tuples in chunks, placing them to fill in the remaining capacity on each cluster node. This entire process, from the moment a load imbalance is detected to when corrective reconfiguration is started, lasts less than twenty seconds.

The E-Store framework has been integrated into the H-Store DBMS [48], a distributed, ACID-compliant DBMS designed for OLTP workloads. E-Store enables H-Store to automatically detect load imbalances, add or remove machines as needed, and redistribute data across the cluster to improve system performance. E-Store successfully satisfies the two major requirements of elasticity for modern OLTP DBMSs: it guarantees ACID properties and performs all of its actions without manual intervention.

1.4 Predictive Modeling for Proactive Elasticity

The E-Store system enables a DBMS to automatically adapt to unpredictable workload changes to meet the throughput and latency requirements of its clients. The performance of E-Store deteriorates during reconfiguration, however, because reconfiguration is only triggered when the system is already under heavy load. These issues could be avoided if reconfiguration were started earlier, but that requires knowledge of the future workload. Fortunately, OLTP workloads often follow a cyclic, predictable pattern. The second system presented in this thesis, P-Store, takes advantage of these patterns to reconfigure the DBMS *before* performance issues arise.

Responding reactively to load changes is not an option for web-based consumer-facing companies because their customers will experience slower response times at the start of a load spike when the database tries to reconfigure the system to meet demand. Many from industry have documented that slow response times for end users leads directly to lost revenue for the company. For example, Amazon found that every 100 ms of increased latency cost them 1% of revenue [58]. Similarly, Google stated that a 500 ms increase in latency caused traffic to drop by 20% [59]. There are many other examples of lost revenue

due to slow response times [15, 55, 16]. In a sense, the start of the overload period is exactly the wrong time to begin a reconfiguration, which is a weakness of all reactive techniques.

P-Store is the first elastic OLTP DBMS to use state-of-the-art time-series prediction techniques to forecast future load on the database. Instead of waiting for the database to become overloaded, it proactively reconfigures the database before the overload occurs. To decide when to start a new reconfiguration and how many machines to allocate at any given time, P-Store uses a novel dynamic programming algorithm. The algorithm produces a schedule that minimizes the number of machines allocated while ensuring sufficient capacity for the predicted load.

To ensure that the algorithm schedules reconfigurations so they complete before an overload happens, P-Store needs an estimate of how long each reconfiguration will take. To create this estimate, P-Store characterizes the time required to execute its novel scheduling algorithm for reconfiguration with minimal performance impact. P-Store also determines the effective capacity of the system during migration, as well as the cost of migration in terms of average machines allocated.

Similar to E-Store, P-Store has been incorporated into the H-Store DBMS. It enables H-Store to proactively scale without manual intervention, once again guaranteeing ACID properties throughout.

1.5 Contributions

This thesis contributes a conceptual framework and system implementation to approach the problem of achieving high throughput and low latency for OLTP workloads in the presence of workload skew and variability. We show that database elasticity is an effective solution, enabling the database to automatically rebalance data and expand and contract resources to adapt to changes in the workload.

The major contributions of this work are:

- E-Store
  - A novel two-tiered partitioning strategy to enable fine-grained mapping of hot tuples and cold chunks of data to servers.

  - A lightweight monitoring component that uses CPU utilization statistics to detect load imbalances and brief periods of detailed monitoring to identify hot tuples.
  - An efficient planning component that uses greedy heuristics to produce near-optimal, two-tiered partition plans.

- P-Store
  - A novel dynamic programming algorithm to determine when and how to reconfigure a database given accurate predictions of future load.
  - A novel scheduling algorithm for executing a reconfiguration, as well as a model characterizing the elapsed time, cost and effective system capacity during the reconfiguration.
  - An analysis showing the effectiveness of using Sparse Periodic Auto-Regression (SPAR) for predicting database workloads.
  - An open-source benchmark to model an online retail application.

- Evaluation
  - A comprehensive evaluation of E-Store on three different benchmarks showing E-Store's ability to manage many types of workload variability and skew. The evaluation demonstrates that under skewed workloads, the E-Store framework improves throughput by up to $4\times$ and reduces query latency by $10\times$.
  - A comprehensive evaluation of P-Store using a real online retail dataset and workload from B2W Digital (B2W). The evaluation shows that P-Store can successfully predict and manage the workload of B2W. It outperforms E-Store on B2W's workload by causing 72% fewer latency violations, and achieves performance comparable to static allocation for peak demand while using 50% fewer resources.

- Limitations
  - Both E-Store and P-Store are designed for partitionable workloads in which transactions mostly access data corresponding to a single partitioning key. This

restriction enables both systems to move data without considering distributed transactions. It limits the applicability of the research for social networks and graphs, but there are numerous other applications, especially those with a customer-centric focus, for which the research is highly relevant.

- E-Store currently supports scaling in and out by only one machine at a time. It can begin a new scale-out operation as soon as it has finished the previous reconfiguration.
- P-Store assumes that load predictions are accurate to within a small error. If the predictions are inaccurate, its performance degrades to that of a reactive system.
- P-Store expects that the workload mix (i.e., types of transactions) and database size are not rapidly changing. Gradual changes can be handled by re-discovering parameters of the model.
- B2W uses the Riak DBMS [56] for their production workload with a cluster of several servers, but since H-Store is a much faster system than Riak, the workload can be managed by a single H-Store server. To demonstrate the benefits of elasticity with the B2W workload, we had to add a small delay to each H-Store transaction so that multiple servers were necessary.
- P-Store is designed for relatively uniform workloads and data distributions. The B2W workload has transient periods of skewed access patterns, which cause some performance degradation in P-Store.

1.6 Thesis Overview

The rest of this thesis is organized as follows:

Chapter 2 will provide background needed to understand the remainder of the thesis. It will explain the key components of the elasticity model including monitoring the DBMS, determining when and how to reconfigure the database, and performing live reconfiguration of the database. Chapter 2 will also explain the details of H-Store and its live migration system, Squall. H-Store is the OLTP DBMS used in this thesis because it is a highly scalable, main-memory DBMS with a shared nothing, partitioned architecture. Most importantly, its

live migration system, Squall, enables fine-grained partitioning and reconfiguration while keeping the system transactionally consistent and highly available.

Chapter 3 will describe the E-Store system in detail. The key feature of E-Store is a two-tiered partitioning scheme to manage "hot tuples" separately from cold tuples. The end-to-end framework of E-Store starts with a two-phase monitoring component, which detects load imbalances by monitoring CPU utilization, and in case of a load imbalance, activates tuple-level monitoring to detect hot tuples. The planning component uses a two-tiered greedy planning algorithm to re-partition the database. Once planning is done, Squall reconfigures the database by offloading the hottest partitions first. The evaluation shows that the two-tiered greedy planning algorithm is superior to one-tiered algorithms and performs as well as computationally intensive approaches on three different benchmarks.

Chapter 4 will describe the P-Store system in detail. The key feature of P-Store is its use of predictive modeling to determine when and how to reconfigure the database. The end-to-end framework of P-Store starts with a monitoring and prediction component, which tracks historical load on the database and uses a predictive model to forecast future load. Then a planning algorithm uses this load prediction to determine when to reconfigure the database so that the average number of machines is minimized, but there is always sufficient capacity for the predicted load. Finally, Squall reconfigures the database in the most efficient way possible that does not overload any partition. The evaluation shows that P-Store outperforms reactive techniques on B2W Digital's real database workload, and saves 50% of computing resources compared to peak provisioning.

Chapter 5 will discuss related work. There are many other elasticity techniques that have been studied, but none use E-Store's highly effective two-tiered approach for managing skew. Many stateless systems have used predictive modeling for elastic scaling, but P-Store is the first OLTP DBMS to use prediction for this purpose. Chapter 5 also examines existing live migration techniques for VMs and databases. Both P-Store and E-Store have used migration scheduling for Squall in a novel way to improve performance. P-Store has gone a step further to characterize the elapsed time, cost, and system capacity during reconfiguration, which has not been done in other systems.

Chapter 6 will discuss ideas for future directions. There are many possible extensions

to E-Store and P-Store, and an obvious direction is to unify the systems into a single system that can manage skew and scale proactively. Another direction for future work is to combine elasticity with replication and try to improve performance by varying the number of replicas for each tuple. Yet another direction is to compare the "scale out" approach to elasticity with the "scale up" approach, and understand whether the best approach is workload-dependent. Beyond OLTP, there are many directions for future work in elasticity for analytic workloads. Even beyond elasticity, there are other ways to make DBMSs adaptable, and future work should investigate new methods for adaptation.

Finally, Chapter 7 will conclude with a summary of the research.

Chapter 2

Background

This chapter introduces the elasticity model and discusses the applicability of the ideas in this thesis. Next, it provides an overview of the underlying architecture of H-Store [54], the multi-node, shared nothing main-memory OLTP DBMS used by the systems presented in this thesis. Finally, it describes Squall [28], the system used by H-Store for live migration of data.

2.1 Elasticity Model

As described in Chapter 1, many new OLTP DBMSs have adopted a shared nothing, partitioned architecture to achieve superior scalability and performance. As such, shared nothing, partitioned DBMSs are the focus of the elasticity model in this thesis. Figure 2-1 shows a schematic of this architecture and the path of a sample transaction. The five steps along the path are: (1) A client sends a query to the DBMS to retrieve data for the employee with $ID = 2$. (2) A transaction manager looks up the partitioning key $ID = 2$ in a table to find which server contains the data. (3) The transaction manager finds that the data resides on Server 1, so it forwards the query to that server. (4) Server 1 executes the query on its local partition of the `EMPLOYEES` table. (5) The DBMS returns the result, which consists of a single row including the ID and the employee's name, John.

This example demonstrates one of the key advantages of the shared nothing, partitioned architecture: since Server 1 contains all of the data needed by the query, no communication

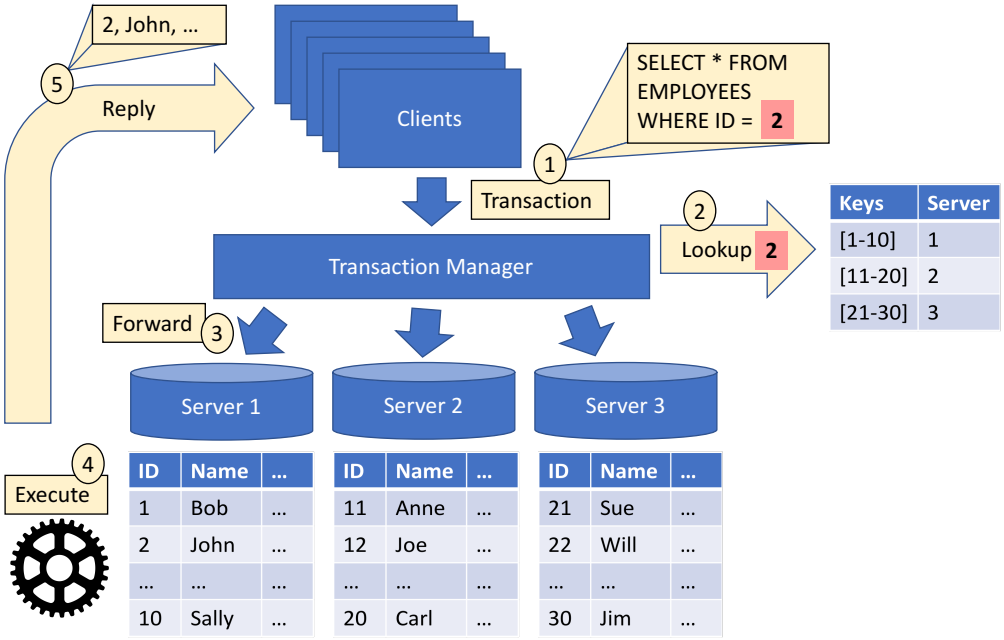


Figure 2-1: Schematic of a shared nothing, partitioned DBMS

with other servers is necessary. If other clients simultaneously issue similar transactions for data on other servers, the transactions can be executed in an "embarrassingly parallel" fashion. A downside of this architecture is that it is highly sensitive to changes in access patterns. For example, if some of the data on Server 1 becomes "hot", Server 1 might become overloaded and start to perform poorly. Because Servers 2 and 3 do not have access to the hot data, they will be mostly idle. A shared storage architecture in which all three servers have access to all data could alleviate some of the issues of skew for a read-only workload, but this architecture is still susceptible to skew for a workload with writes. This susceptibility is due to the fact that despite the shared storage layer, data accesses are typically still partitioned between servers at the computation layer to avoid concurrency issues and cache misses. Regardless of the architecture, if the aggregate load increases beyond what three servers can handle, all servers will become overloaded even if there is no skew.

In order for a shared nothing, partitioned DBMS to be resilient to changes in access patterns and aggregate load, it requires a mechanism for elasticity which can perform the following two actions:

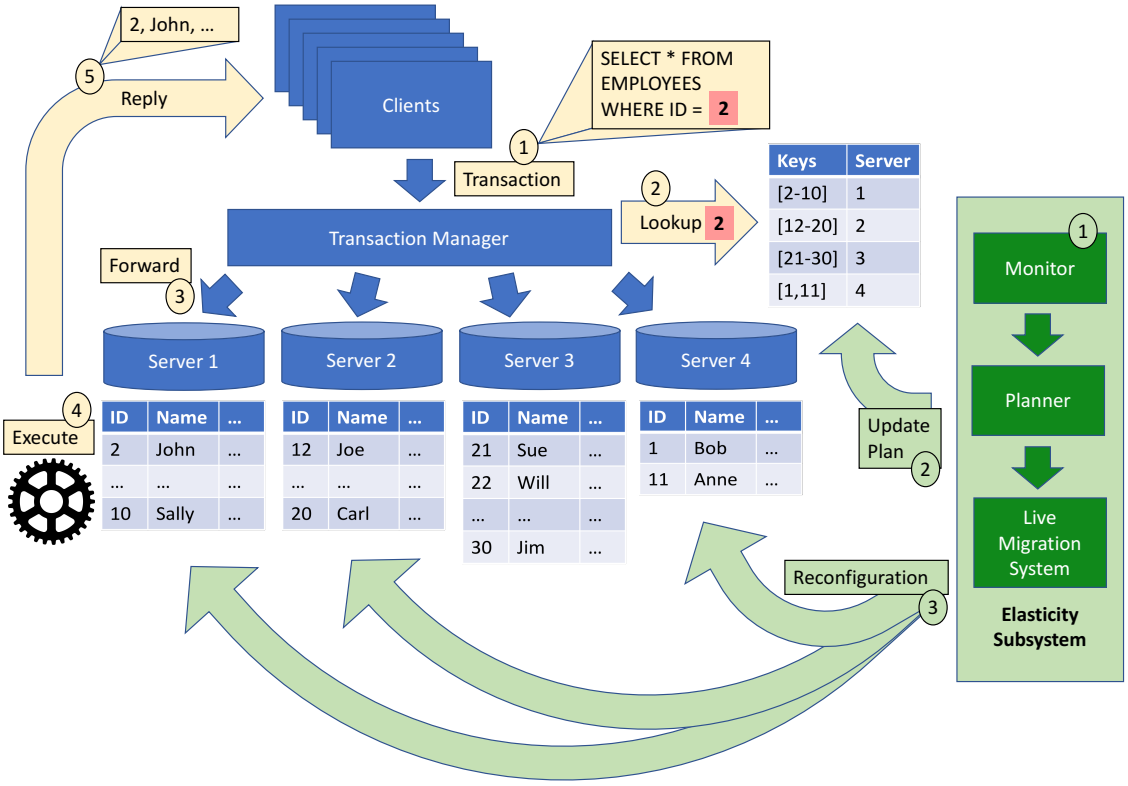


Figure 2-2: Schematic of a shared nothing, partitioned DBMS with an elasticity subsystem

1. Add or remove servers as needed to manage an increase or decrease in aggregate load
2. Move data between servers by repartitioning (resharding) the database to balance load

. Figure 2-2 shows a schematic of the same database from Figure 2-1, but with the addition of an elasticity subsystem. The high level elasticity model in this thesis has three major components: a Monitor, a Planner, and a Live Migration System. We will return to this figure in Chapters 3 and 4 to discuss how E-Store and P-Store each implement and extend this model. (1) The Monitor is responsible for collecting data as the DBMS runs, and triggering a reconfiguration if certain conditions are met. For example, the monitoring component in E-Store triggers a reconfiguration if CPU utilization falls above or below certain thresholds for a specified period of time. (2) Once reconfiguration is triggered, the Monitor sends data to the Planner, which is responsible for updating the partition plan according to the policies of the specific elasticity system. In the example shown, the Planner updates the partition plan so that keys 1 and 11 are moved to Server 4. (3) The Planner sends

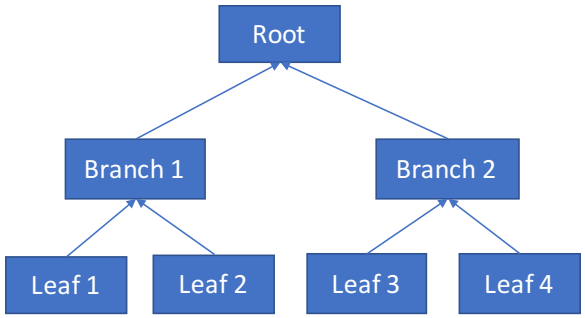


Figure 2-3: A tree-based schema in which all non-replicated tables in the database are connected to the root table via primary-key-foreign-key relationships

the new plan to the Live Migration System, which is responsible for reconfiguring the data in the database to match the new partition plan. In this example, the Live Migration System adds a fourth server and moves Bob and Anne (IDs 1 and 11) to the new server.

The shared nothing, partitioned architecture is ideal for workloads whose transactions access data at a single node, but workloads with distributed transactions, i.e. transactions that span multiple nodes, execute slower [76]. Obviously any data rearrangement by an elastic DBMS could change the number of multi-node transactions. Therefore, the Planner component of these systems must consider what data elements are accessed together by transactions when making decisions about data placement and load balancing. This presents a complex optimization environment. Hence, this thesis focuses on an important subset of the general case. Specifically, it assumes all non-replicated tables of an OLTP database form a tree-schema based on foreign key relationships (see Figure 2-3). Although this rules out graph-structured schemas (such as social networks) and *m-n* relationships, it applies to many real-world OLTP applications [89]. For example, any customer-centric application such as an online retail or commercial banking application will be easily partitionable by customer ID.

A straightforward physical design for tree schemas is to partition tuples in the root node and then co-locate every descendant tuple with its parent tuple. This *co-location* tuple allocation strategy is the best strategy as long as the majority of transactions access the schema tree via the root node and descend some portion of it during execution by following foreign key relationships. Consistent with the tree metaphor, this access pattern follows *root-to-leaf order*. For example, in the popular OLTP database benchmark TPC-C [96], tuples of all

non-replicated tables have a foreign key identifier that refers to a tuple in the WAREHOUSE table. Moreover, 90% of the transactions access the database in root-to-leaf order. As a result, partitioning tables based on their WAREHOUSE id and co-locating descendant tuples with their parent minimizes the number of multi-partition transactions.

This thesis assumes that the DBMS starts with a co-location allocation, and the problem for the elasticity planner is to find a second co-location allocation that balances the load and does not overload nodes. Distributed transactions are not considered as part of the planning process.

2.2 H-Store System Architecture

The elasticity techniques discussed in this thesis are generic and can be adapted to any shared-nothing, partitioned DBMS for workloads with tree structured schemas. H-Store is the DBMS used by the initial prototypes of E-Store and P-Store because it is an ACID-compliant DBMS with a shared nothing, partitioned architecture, and it has a live migration system for reconfiguring the database. This section provides more background about H-Store, and the following section describes its live migration system, Squall.

H-Store is a distributed, main-memory DBMS that runs on a cluster of shared-nothing compute nodes [54]. Figure 2-4 illustrates the H-Store architecture. An H-Store instance is defined as a cluster of two or more nodes deployed within the same administrative domain. A *node* (also called a *server*) is a single physical machine that manages one or more logical data partitions.

Each partition is assigned to a single-threaded *execution engine* that has exclusive access to the data at that partition. This engine is assigned to a single CPU core in its host node. The single-threaded nature of the execution engine means that transactions accessing a single partition do not require any locks or latches, making execution extremely efficient. When a transaction finishes execution, the engine can work on another transaction. Each node also contains a coordinator that allows its engines to communicate with the engines on other nodes.

H-Store supports ad-hoc queries but it is optimized to execute transactions as stored

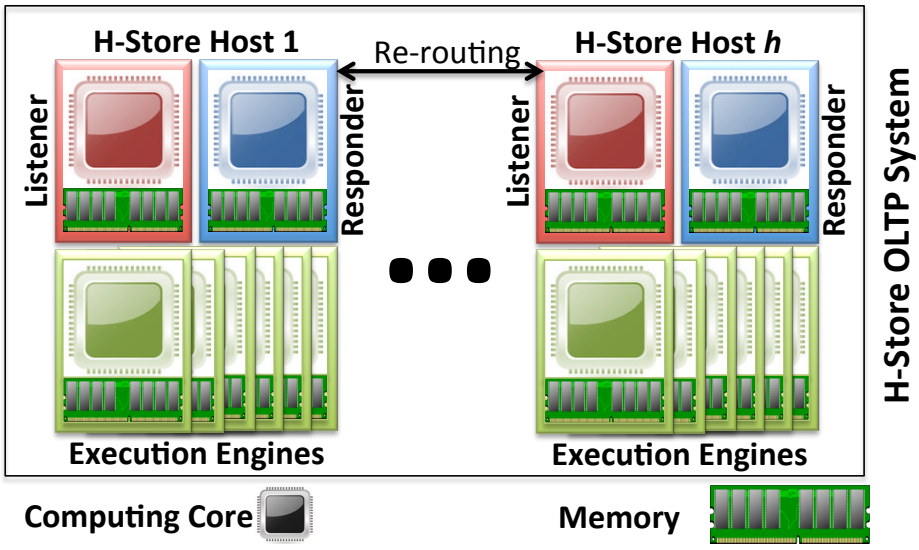


Figure 2-4: The H-Store Architecture.

procedures. This thesis uses the term *transaction* to refer to an invocation of a stored procedure. A stored procedure contains *control code* (i.e., application logic) that invokes pre-defined parameterized SQL commands. A client application initiates a transaction by sending a request (a stored procedure name and input parameters) to any node. Each transaction is *routed* to one or more partitions and their corresponding servers that contain the data accessed by a transaction.

H-Store supports *replicating* tables on all servers, which is particularly useful for small read-only tables. This work, however, focuses on horizontally partitioned tables, where the tuples of each table are split into disjoint sets and allocated without redundancy to the various nodes managed by H-Store. The assignment of tuples to partitions is determined by one or more columns, which constitute the *partitioning key*, and the values of these columns are mapped to partitions using either range- or hash-partitioning. Transactions are thus routed to specific partitions based on the set of partitioning keys they access. Most tables in the workloads we study have only one partitioning attribute, but H-Store supports partitioning based on an arbitrary number of columns.

H-Store can run at close to the speed of main memory as long as data is not heavily skewed, there are few distributed transactions, and there are enough CPU cores and data partitions to handle the incoming requests. This thesis will show how scaling out and

```
plan:{
  ``warehouse (W_ID)'': {
   ``Partition 1'' : [0-3)
   ``Partition 2'' : [3-5)
   ``Partition 3'' : [5-9)
   ``Partition 4'' : [9-10)
}}
```

(a) Old Plan

```
plan:{
 ``warehouse (W_ID)'': {
  ``Partition 1'': [0-2)
  ``Partition 2'': [3-5)
  ``Partition 3'': [2-3),[5-6)
  ``Partition 4'': [6-10)
}}
```

(b) New Plan

Figure 2-5: An example of an updated partition plan for a TPC-C database.

reconfiguring H-Store with a live migration system can help alleviate performance issues due to skew and heavy loads. Prior work has also shown how to alleviate performance issues due to distributed transactions [83].

Although the techniques discussed in this thesis are implemented for H-Store, they are generic and can be adapted to other shared-nothing DBMSs that use horizontal partitioning, whether or not the interface is through stored procedures. H-Store's speculative execution facilities [77] are competitive with other concurrency control schemes and its command logging system has been shown to be superior to data logging schemes [62].

2.3 The Squall Live Migration System

To elastically scale a shared nothing system such as H-Store, it is necessary to move data between nodes. Squall [28] has been built into H-Store for this purpose, and it is the data migration system used by both E-Store and P-Store. Unlike some other DBMSs, H-Store is designed for OLTP workloads which require strong consistency and high availability. Therefore, when migrating data, Squall must ensure transactional consistency while also keeping the system live and available to process transactions with minimal overhead.

Squall supports fine-grained reconfiguration with flexible range partitioning, in which any number of ranges may be mapped to a single partition, and ranges may be of arbitrary size. This scheme is ideal for elasticity because it allows for maximum flexibility when assigning tuples to partitions. For example, E-Store assigns hot tuples to specific partitions

using ranges of size one. A *partition plan* is used to define the partitioning scheme of the database, and can be represented with a json file mapping ranges of keys to partitions. When E-Store and P-Store determine that a reconfiguration is required for elastic scaling or load balancing, they start from the current partition plan and determine a new plan with the desired configuration (the details of this planning process will be discussed later). An example of such a transformation for a TPC-C database with ten warehouses is shown in Figure 2-5. Given the new plan, Squall is responsible for physically moving data so that the storage layout of the database matches the new plan.

To execute reconfiguration, Squall proceeds though three stages: initialization, migration, and termination. Let us examine each phase in detail.

2.3.1 Initialization

There is one distributed transaction across all nodes to start the reconfiguration, so that all servers are simultaneously aware of the new partition plan and that reconfiguration is under way. One node is designated as the "leader". After the initial distributed transaction, initialization proceeds in a decentralized manner. Given the old and new plans, each node individually determines which ranges of keys are moving in or out. This can be done by essentially performing a "diff" between the two plans. A single partition may be a *source* for some ranges that are moving out, and a *destination* for other ranges that are moving in. In Figure 2-5, for example, partition 3 is both a source and a destination for different ranges. After determining the incoming and outgoing ranges for its local partitions, each node creates data structures to track the progress of these ranges. Initially, each range is labeled as "not started".

2.3.2 Migration

After initialization, each destination partition begins to asynchronously issue pull requests for its incoming ranges to the source partition(s). When a source partition receives a pull request, it extracts the requested data and sends it to the destination. To ensure transactional consistency, Squall conforms to H-Store's single-threaded model of execution: while data

is being extracted from the source partition, no other transactions can be executed on that partition. Similarly, while data is being loaded at the destination, no transactions can be executed there. Therefore, moving large amounts of data may cause transactions to be blocked for a long period of time. To prevent performance issues that could be caused by large migrations, Squall splits up large ranges into smaller chunks, and spaces them apart in time. Interleaving transactions between pulls is what allows Squall to complete reconfiguration with minimal performance impact. The size of each chunk and the amount of time between pulls can be tuned based on the application's tolerance to performance degradation and desired total reconfiguration time.

As soon as a range has begun to move, it is marked "partial" in the data structures of the source and destination nodes. When the last chunk has been sent or received for a given range, the range is marked "complete". If a transaction arrives at a partition and attempts to access an incoming range that has been marked "not started" or "partial", a blocking request is issued to the source partition to extract the remaining data so the transaction can execute on the destination partition. When all incoming ranges for a node's local partitions have been marked "complete", the node notifies the leader that it has received all of the data it is expecting.

Squall does not by default specify an order for the pull requests, so both E-Store and P-Store have extended Squall to enable deliberate scheduling of reconfigurations. For example, a major goal of E-Store is to reduce hot spots, so it instructs Squall to prioritize data movements that offload data from hot partitions to cold partitions. One of P-Store's goals is to eliminate the performance impact of reconfiguration, so it schedules data movements in such a way that reconfiguration completes as fast as possible while never overloading a single partition. The details of these scheduling algorithms will be discussed in Chapters 3 and 4.

2.3.3 Termination

Once all nodes have notified the leader that they have received all expected data, the leader broadcasts a notification that reconfiguration has completed. Each node deletes the data

structures that were set up to track reconfiguration, and returns to normal execution.

2.4 Summary

This chapter has described the elasticity model of this thesis, which consists of three major components: a Monitor, a Planner, and a Live Migration System. The elasticity model is designed for OLTP DBMSs with a shared nothing, partitioned architecture, and for workloads with a tree-structured schema. The two elasticity systems presented in this thesis have been integrated into H-Store, a main-memory, shared nothing DBMS which is highly scalable for partitionable workloads. H-Store is also ideal for elasticity because of its live migration system, Squall. Squall supports fine-grained partitioning and reconfiguration, and reconfigures the database with minimal performance disruption.

Chapter 3

E-Store

E-Store is an elastic database system designed to adapt to workload changes, with a special focus on managing skew using a *two-tiered* partitioning scheme. This chapter describes the E-Store system in detail, including its components for monitoring load on the database and detecting hot spots, algorithms for repartitioning the database to achieve load balancing and scalability, and interactions with the live migration system for executing the reconfiguration. An evaluation of E-Store on three different workloads shows its ability handle many different types of variability and skew, improving throughput by up to 4× and reducing latency by 10×.

Figure 3-1 shows E-Store's major extensions to the model introduced in Chapter 2. As shown in the orange boxes, the key differentiators of E-Store are a *hot tuples detector* in the monitoring component, *two-tiered planning* algorithms for repartitioning the database, and a live migration scheduler which *offloads hot partitions first*.

3.1 Motivation

To illustrate the impact of skew on an OLTP DBMS, we conducted an initial experiment using the Yahoo! Cloud Serving Benchmark (YCSB) [19] on a five-node H-Store cluster. YCSB is a workload designed to test key-value data stores, and consists mostly of single-key reads and updates on a single table. For this setup, we used a database with 60 million tuples that are each 1KB in size (∼60GB in total) that are deployed on 30 par-

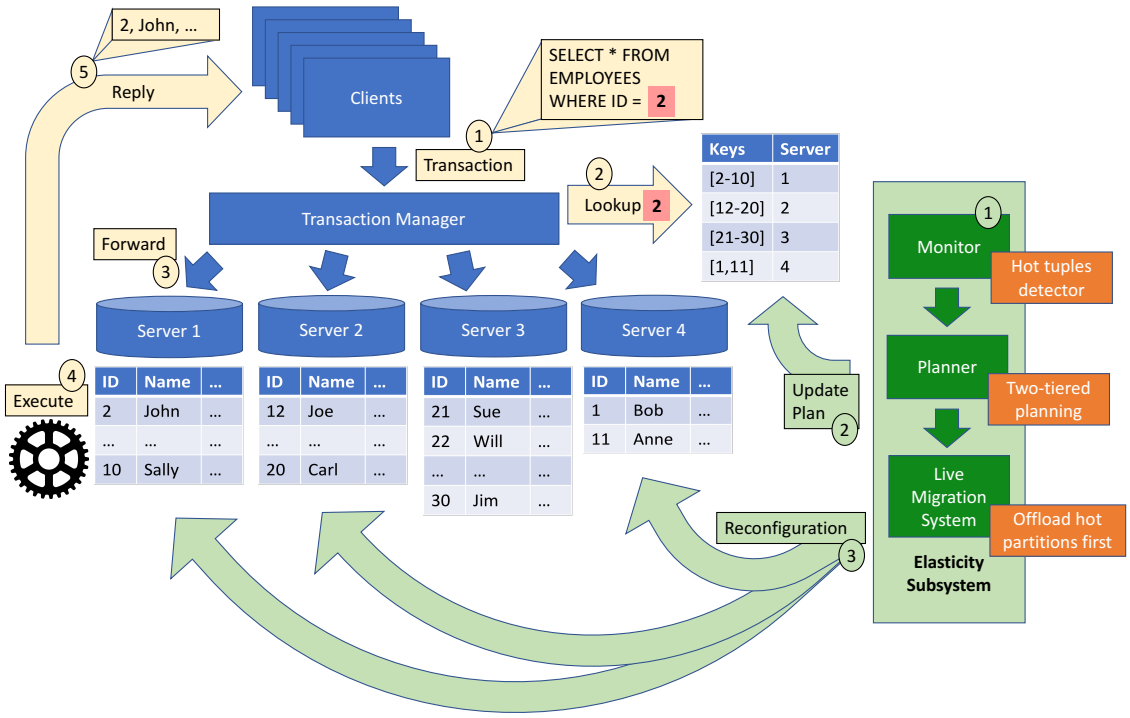


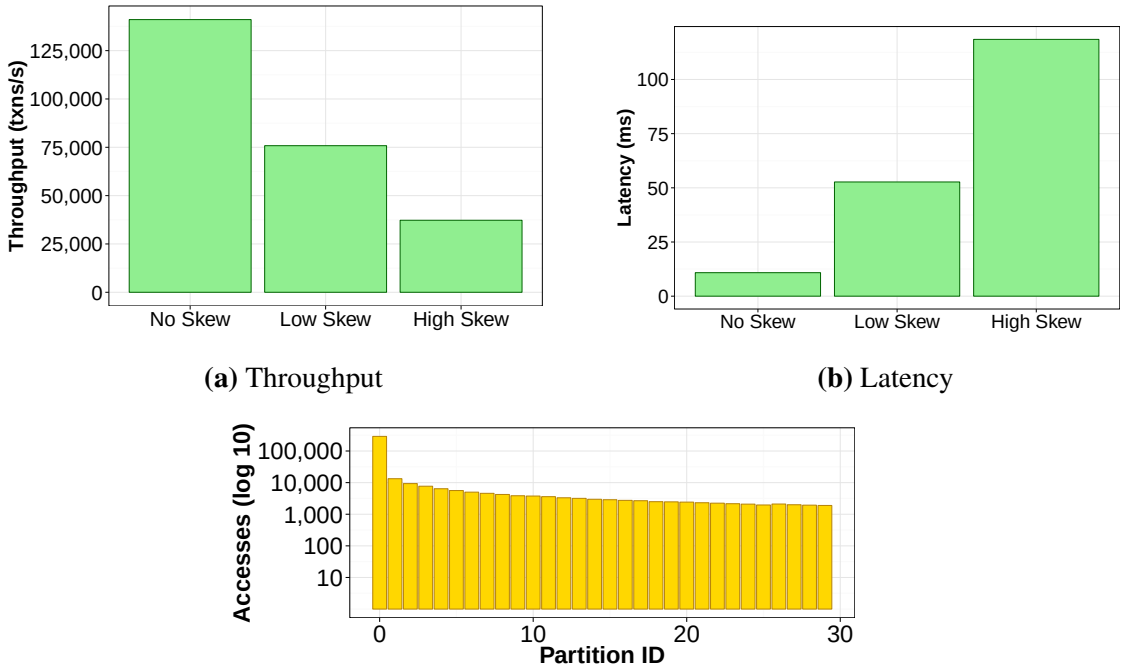
Figure 3-1: Schematic of a shared nothing, partitioned DBMS with an elasticity subsystem. Special components of E-Store are shown in the orange boxes.

titions (six per node). Additional details of the evaluation environment are described in Section 3.5. We modified the YCSB workload generator to issue transaction requests with three access patterns:

1. **No Skew:** A baseline uniform distribution.
2. **Low Skew:** A Zipfian distribution where two-thirds of the accesses go to one-third of the tuples.
3. **High Skew:** The above Zipfian distribution applied to 40% of the accesses, combined with additional "hotspots", where the remaining 60% of the accesses go to 40 individual tuples in partition 0.

For each skew pattern, we ran the workload for ten minutes and report the average throughput and latency of the system. We also collected the average CPU utilization per partition in the cluster.

We see in Figure 3-2 that the DBMS's performance degrades as the amount of skew in the workload increases: Figure 3-2a shows that throughput decreases by 4× from the



(a) Throughput

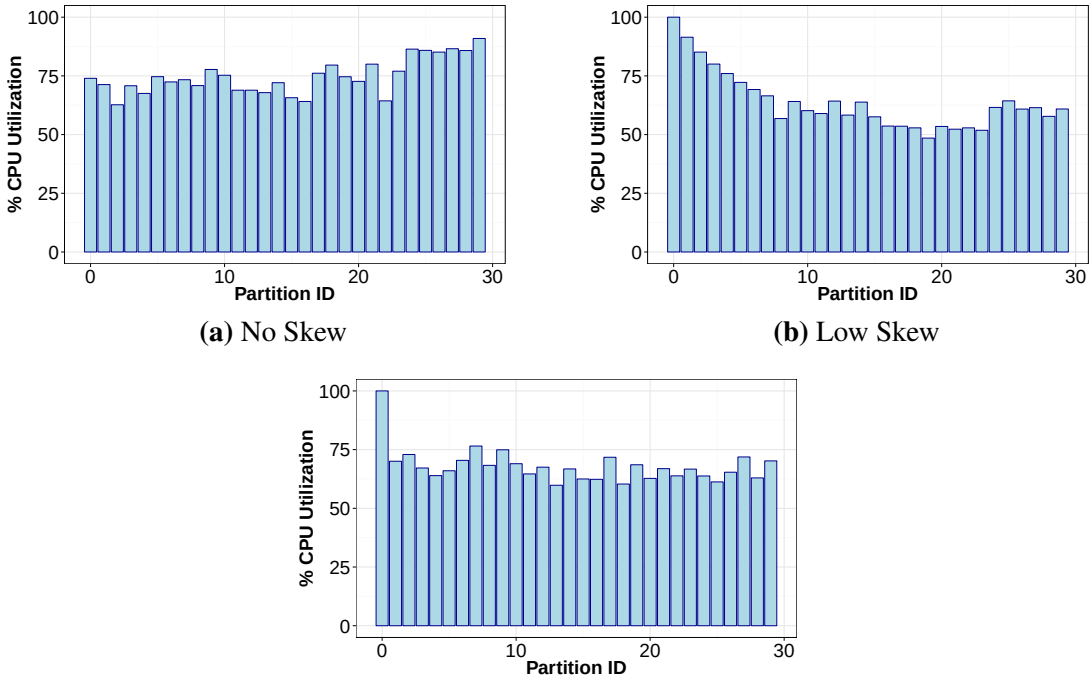
(b) Latency

(c) Tuple Accesses per Partition with High Skew

Figure 3-2: Latency and throughput measurements for different YCSB workloads with varying amounts of skew. In Figure 3-2c, we show the total tuple accesses per partition over a 10 second window for the **high skew** workload.

no-skew to high-skew workload, while Figure 3-2b shows that latency increases by 10×. To help understand why this occurs, the chart in Figure 3-2c shows the number of tuples that were accessed by transactions for the high-skew workload. We see that partition 0 executes an order of magnitude more transactions than the other partitions. This means that the queue for that partition's engine is longer than others causing the higher average latency. Also, other partitions are idle for periods of time, thereby decreasing overall throughput.

This load imbalance is also evident in the CPU utilization of the partitions in the cluster. In Figure 3-3, we see that the variation of CPU utilization among the 30 partitions increases proportionally to the amount of load skew. Again, for the high skew workload in Figure 3-3c, partition 0 has the most utilization because it has the highest load.



(a) No Skew

(b) Low Skew

(c) High Skew

Figure 3-3: Partition CPU utilization for the YCSB workload with varying amounts of skew. The database is split across five nodes, each with six partitions.

3.2 The E-Store Framework

To ensure high performance and availability, a distributed DBMS must react to changes in the workload and dynamically reprovision the database without incurring downtime. This problem can be broken into three parts:

1. How to identify load imbalance requiring data migration?
2. How to choose which data to move and where to place it?
3. How to physically migrate data between partitions?

The E-Store framework shown in Figure 3-4 handles all three issues for OLTP applications. It is comprised of three components that are integrated with the DBMS. To detect load imbalance and identify the data causing it, the *E-Monitor* component communicates with the underlying OLTP DBMS to collect statistics about resource utilization and tuple accesses. This information is then passed to the *E-Planner* to decide whether there is a

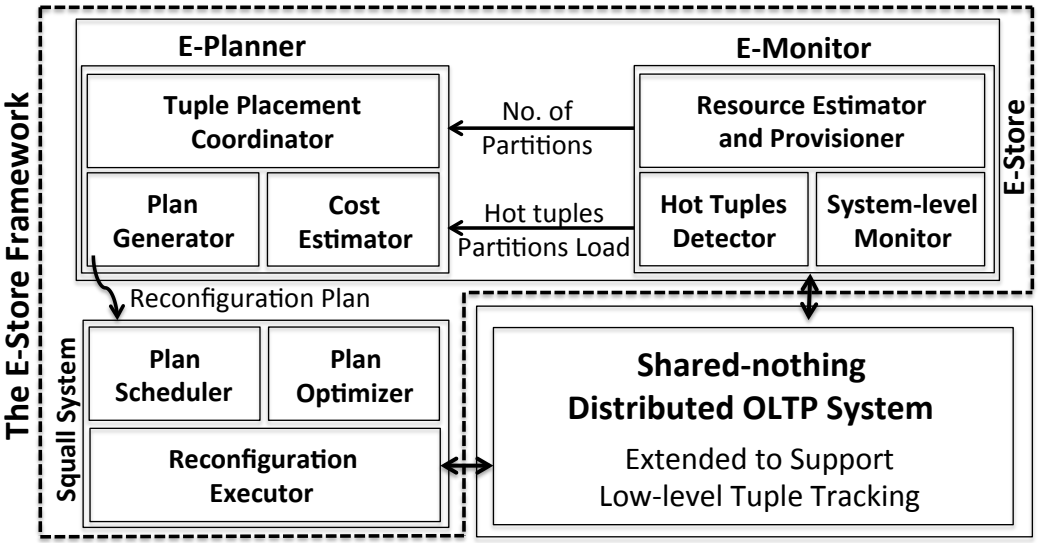


Figure 3-4: The E-Store Architecture.

need to add or remove nodes and/or re-organize the data. The E-Planner generates a reconfiguration plan that seeks to maximize system performance after reconfiguration while also minimizing the total amount of data movement to limit migration overhead. An overview of how E-Monitor and E-Planner work together to rebalance a distributed DBMS is shown in Figure 3-5.

For physically moving data, E-Store leverages Squall, the live migration system for H-Store. As described in Section 2.3, Squall uses the new reconfiguration plan generated by the E-Planner to decide how to physically move the data between partitions while the DBMS continues to execute transactions. This allows the DBMS to remain on-line during the reconfiguration with only minor degradation in performance.

We now describe how E-Store moves data across the cluster during a reorganization and its two-tier partitioning scheme that assigns data to partitions. We then discuss the details of the E-Monitor and E-Planner components in Sections 3.3 and 3.4, respectively.

3.2.1 Data Migration

E-Store uses an updated version of the Squall live migration system described in Section 2.3. As mentioned previously, Squall does not by default specify the order of data migrations. For this reason, Squall has been modified for E-Store to include an optimizer

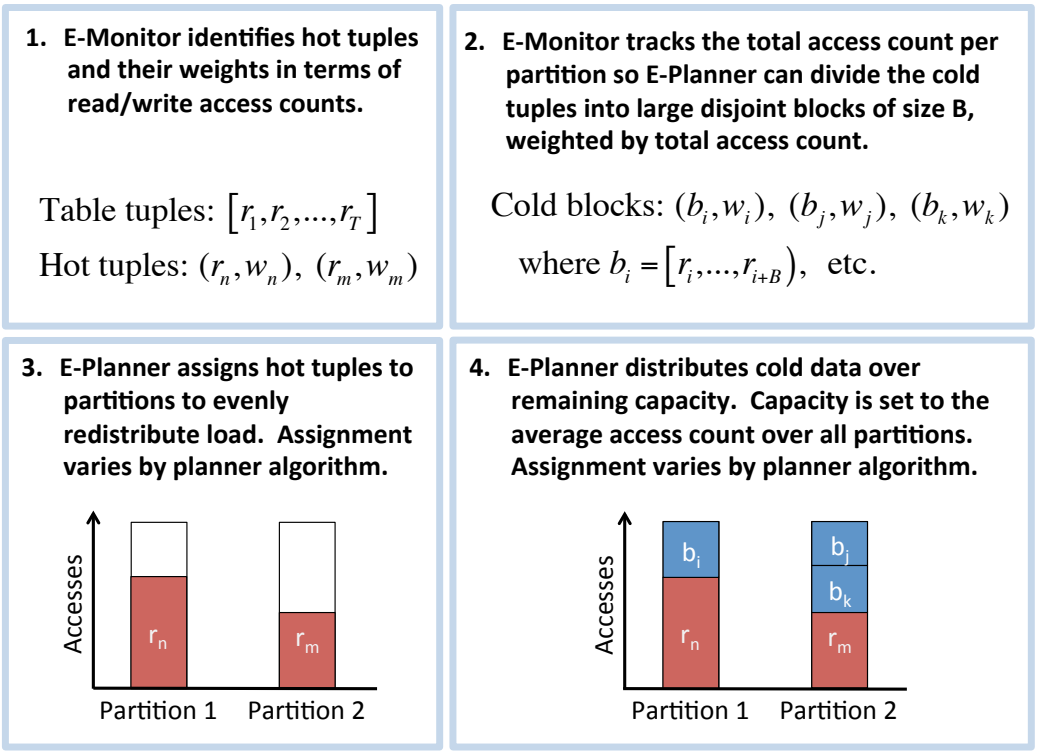


Figure 3-5: The steps of E-Store's migration process.

that decides the order that data is migrated. The optimizer makes the order of migration explicit by splitting the reconfiguration plan into sub-plans, which are executed serially. As shown in the example in Figure 3-6, the plan on the left moves data from partition 1 to partitions 2, 3, and 4. The plan is then divided into three separate sub-plans that each migrate data from partition 1 to just one partition at a time. In the case of applications with many partition keys, such as Voter [90] and YCSB, Squall calculates the ranges of keys that need to be moved and places the ranges that have the same source and destination partitions into the same sub-plan. For applications with fewer unique partition keys, however, this method generates sub-plans that move an excessive amount of data for each key. For example, moving a single `WAREHOUSE` id in the TPC-C benchmark will end up moving many tuples, because as described in Section 2.1, we use a hierarchical co-location strategy for TPC-C to place all tuples of non-replicated tables according to their primary or foreign key `WAREHOUSE` id. In this case, Squall further subdivides single-key ranges by using secondary and tertiary partitioning attributes, thereby limiting the amount of data moved in each sub-plan. For example, the `DISTRICT` id can be used as a secondary parti-

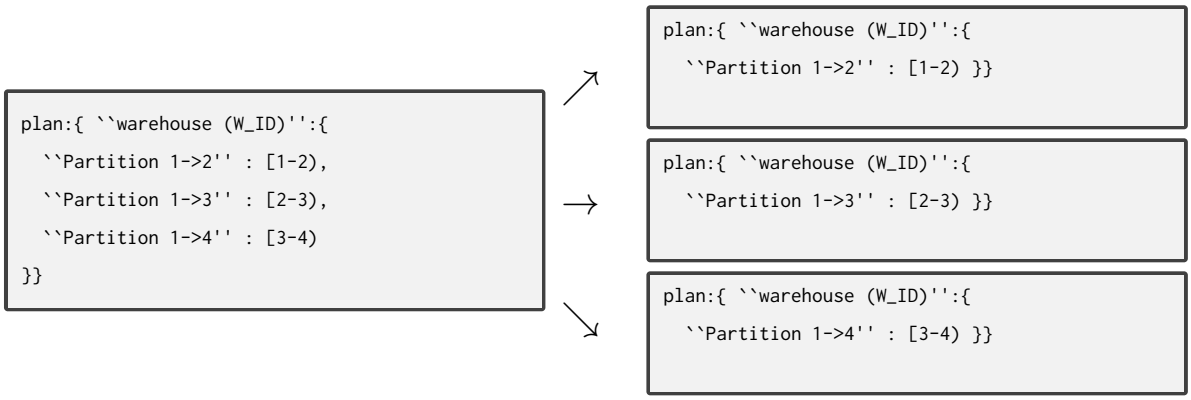


Figure 3-6: A sample reconfiguration plan split into three sub-plans.

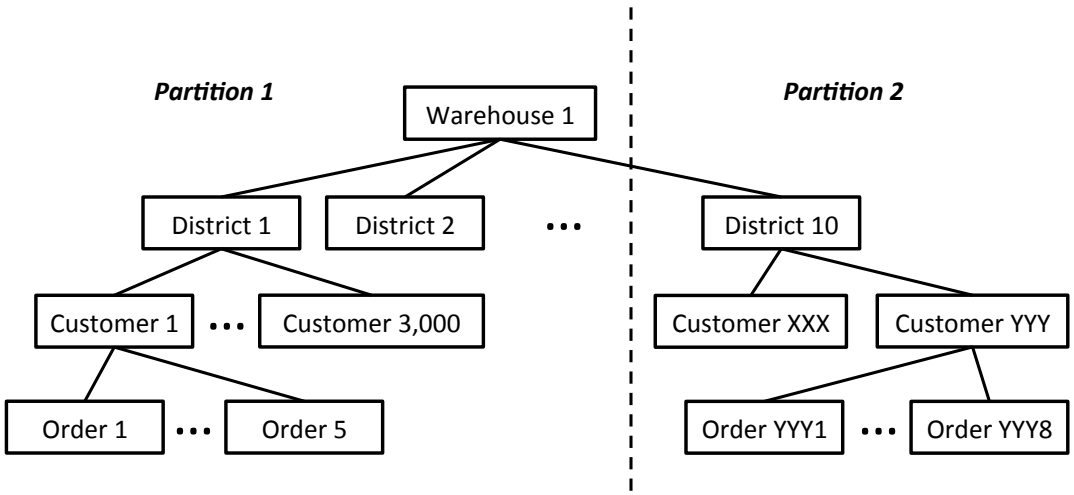


Figure 3-7: During reconfiguration in TPC-C, Squall uses secondary partitioning to split the DISTRICT table to avoid moving an entire WAREHOUSE entity all at once. While migration is in progress, the logical warehouse is split across two partitions, causing some distributed transactions.

tioning attribute for most of the TPC-C tables during migration. Each warehouse contains 10 DISTRICT records, so by partitioning the tables using their DISTRICT ids, Squall can split a warehouse into 10 pieces to limit the overhead of each data pull (see Figure 3-7). Splitting the ranges in this way increases the number of distributed transactions in TPC-C, but avoids blocking execution for extended periods by throttling migrations.

After producing the sub-plans, the optimizer prioritizes them based on which ones send data from the most overloaded partitions to the least overloaded partitions. This splitting ensures that overloaded partitions are relieved as quickly as possible. It also allows periods of idle time to be inserted between the execution of each sub-plan to allow transactions

to be executed without the overhead of Squall's migrations. In this way, any transaction backlog is dissipated. We found that 100 sub-plans provided a good balance between limiting the duration of reconfiguration and limiting performance degradation for all of the workloads that we evaluated. In general, this number is workload-dependent and should be tuned based on the desired duration of reconfiguration and tolerance for performance degradation. For example, larger databases requiring significantly more data to be moved during reconfiguration would likely require more sub-plans.

To execute a sub-plan, the leader first checks whether there is an ongoing reconfiguration. If not, it atomically initializes all partitions with the new plan. Each partition then switches into a special mode to manage the migration of tuples while also ensuring the correct execution of transactions during reconfiguration. During the migration, transactions may access data that is being moved. When a transaction (or local portion of a multi-partition transaction) arrives at a node, Squall checks whether it will access data that is moving in the current sub-plan. If the data is not local, then the transaction is routed to the destination partition or is restarted as a distributed transaction if the data resides on multiple partitions.

3.2.2 Two-Tiered Partitioning

Most distributed DBMSs use a *single-level* partitioning scheme whereby records in a data set are hash partitioned or range partitioned on a collection of keys [76, 20]. This approach cannot handle fine-grained hot spots, such as the NYSE example from Section 1.2. If two heavily traded stocks hash to the same partition, it will be difficult to put them on separate nodes. Range partitioning also may not perform well since those two hot records could be near each other in the sort order for range-partitioned keys. One could rely on a human to manually assign tuples to partitions, but identifying and correcting such scenarios in a timely manner is non-trivial [33].

To deal with such hot spots, E-Store uses a *two-tiered* partitioning scheme. It starts with an initial layout whereby root-level keys are range partitioned into blocks of size B and co-located with descendant tuples. We found that a block size of $B = 100,000$ keys

worked well for a variety of workloads, and that is what we used in the Voter and YCSB experiments in this chapter. For TPC-C, which has only a few root keys, we set $B = 1$. In general, B is somewhat workload-dependent and varies based on the number of distinct root keys. Ideally, B should be less than 1% of the total number of distinct root keys, but not so small that the key ranges become excessively fragmented over time.

Given this initial partitioning of keys, E-Store identifies a collection of k keys with high activity, where k is a user-defined parameter. For most workloads we found that setting k to the top 1% of keys accessed during a specified time window produced good results, as discussed in Section 3.5.2. These keys are extracted from their blocks and allocated to nodes individually. In short, we partition hot keys separately from cold ranges. The framework is illustrated in Figure 3-5. While this approach works well with any number of root-level keys, workloads with a large number of root-level keys will benefit the most. Thus, our two-tiered partitioning scheme is more flexible than previous one-tiered approaches because it accommodates both hot keys and cold ranges.

3.3 Adaptive Partition Monitoring

In order for E-Store's reorganization to be effective, it must know when the DBMS's performance becomes unbalanced due to hotspots, skew, or excessive load. The framework must also be able to identify the individual tuples that are causing hotspots so that it can update the database's two-tier partitioning scheme.

A major challenge in continuous monitoring for high-performance OLTP DBMSs is the overhead of collecting and processing monitoring data. The system could examine transactions' access patterns based on recent log activity [87], but the delay from this off-line analysis would impact the timeliness of corrective action [33]. To eliminate this delay the system could monitor the usage of individual tuples in every transaction, but this level of monitoring is expensive and would significantly slow down execution.

To avoid this problem, E-Store uses a two-phase monitoring component called the *E-Monitor*. As shown in Figure 3-4, E-Monitor is a standalone program running continuously outside of the DBMS. During normal operation, the system collects a small

amount of data from each DBMS node using non-intrusive OS-level statistics such as CPU utilization [53]. Once an imbalance is detected, E-Monitor triggers per-tuple monitoring that is implemented directly inside of the DBMS. After a brief collection period, E-Monitor switches back to lightweight mode and sends the data collected during this phase to E-Planner to generate a migration plan for the DBMS. We now describe these two monitoring phases in more detail.

3.3.1 Phase 1: Collecting System Level Metrics

In the first phase, E-Monitor periodically collects OS-level metrics of the CPU utilization for each partition on the DBMS's nodes. Such coarse-grained, high-level information about the system is inexpensive to obtain and still provides enough actionable data. Using CPU utilization in a main memory DBMS provides a good approximation of the system's overall performance. However, monitoring adherence to service-level agreements (e.g., latency thresholds) [29] would provide a better idea of application performance, and we are considering adding support for this in E-Store as future work.

When E-Monitor polls a node, it retrieves the current utilization for all of the partitions at that node and computes the moving average over the last 60 seconds. E-Monitor uses two thresholds, a *high-* and *low-watermark*, to control whether corrective action is needed. These thresholds are set by the database administrator based on a trade-off between system response time and the desired resource utilization level. If the average utilization across the whole cluster goes below the low-watermark, E-Store will choose to take the most under-utilized node off-line. If the high-watermark is exceeded by at least one partition, the system should balance load and add more servers, if needed. If a watermark is exceeded, E-Monitor triggers a phase of more detailed tuple-level monitoring.

For our experiments in Section 3.5, we configured the system to check each node every five seconds; retrieving utilization data more often than this did not make a significant difference in how quickly E-Store was able to respond to imbalance. We set the high-watermark to 90% to leave some headroom for sudden load spikes. Likewise, we set the low-watermark to 50% to avoid scaling in too eagerly.

3.3.2 Phase 2: Tuple-Level Monitoring

Once E-Monitor detects an imbalance, it starts the second phase of tuple-level monitoring on the entire cluster for a short period of time. The framework gathers information on the hot spots causing the imbalance to determine how best to redistribute data. Since E-Store focuses on tree-structured schemas and their co-location strategies (see Section 2.1), monitoring only the root tuples provides a good approximation of system activity and minimizes the overhead of this phase.

We define the hot tuples to be the top-k most frequently accessed tuples within the time window W. A tuple is counted as "accessed" if it is read, modified, or inserted by a transaction. For this discussion, let $\{r_1, r_2, \ldots, r_m\}$ be the set of all tuples (records) in the database and $\{p_1, p_2, \ldots, p_c\}$ be the set of partitions. For a partition p_j, let $L(p_j)$ denote the sum of tuple accesses for that partition and $TK(p_j)$ denote the set of the top-k most frequently accessed tuples. Thus, a tuple r_i is deemed "hot" if $r_i \in TK$. For convenience, we have included these and other symbols used throughout the chapter in a table in Appendix A.

When tuple-level monitoring is enabled, the DBMS initializes an internal histogram at each partition that maps a tuple's unique identifier to the number of times a transaction accessed that tuple. After the time window W has elapsed, the execution engine at each node assembles L and TK for its local partitions and sends them to E-Monitor. Once E-Monitor receives this information from all partitions, it generates a global top-k list. This list is used by E-Store's reprovisioning algorithms to build a reconfiguration plan. This monitoring process enables E-Monitor to collect statistics on all root-level tuples. The accesses that do not correspond to top-k tuples are aggregated to obtain access frequencies for the "cold blocks" of tuples.

The database administrator should configure the monitoring time window for this phase to be the shortest amount of time needed to find hot tuples. The optimal value for W depends on the transaction rate and the access pattern distribution. Likewise, it is important to choose the right size for k so that enough tuples are identified as "hot." There is a trade-off between the accuracy in hot spot detection versus the additional overhead on an already overloaded system. We analyze the sensitivity of E-Store to both parameters

in Section 3.5.2.

3.4 Reprovisioning Algorithms

After E-Monitor collects tuple-level access counts, E-Planner uses this data to generate a new partitioning scheme for the database. We now discuss several algorithms for automatically generating a two-level partitioning scheme. We first discuss how E-Planner decides whether to increase or decrease the number of nodes in the cluster. We then describe several strategies for generating new reconfiguration plans to reorganize the database.

All of the reconfiguration plans generated by E-Planner's algorithms begin by promoting any tuples that were newly identified as hot from block allocation to individual placement. Likewise, any tuple that was previously hot but is now identified as cold is demoted to the block allocation scheme. Then the new top-k hot tuples are allocated to nodes. Moving individual tuples between nodes requires little network bandwidth and can quickly alleviate load imbalances, so E-Planner performs these allocations first. If there is still a predicted load imbalance, E-Planner allocates cold blocks as a final step.

Our reprovisioning algorithms currently do not take the amount of main memory into account when producing reconfiguration plans. Adapting to database size changes is left as future work.

3.4.1 Scaling Cluster Size Up/Down

Before starting the reprovisioning process, E-Planner determines whether to maintain the DBMS's present cluster size or whether to add or remove nodes. It makes this decision by using the CPU utilization metrics collected during monitoring. If the average CPU utilization across the whole cluster exceeds the high-watermark, the framework allocates new partitions. In the same way, if the average utilization is less than the low-watermark, it will decommission partitions. E-Store currently only supports changing the DBMS's cluster size by a single node for each reconfiguration round.

3.4.2 Optimal Placement

We developed two different reprovisioning strategies derived from the well-known "bin packing" algorithm. Both of these approaches use an integer programming model to generate the optimal assignment of tuples to partitions. As the evaluation will show, these algorithms take over 20 hours to run, so they are not practical for real-world deployments. Instead, they provide a baseline with which to compare the faster approximation algorithms that we present in the subsequent section.

We now describe our first bin packing algorithm that generates a two-tier placement where individual hot tuples are assigned to specific partitions and the rest of the "cold" data is assigned to partitions in blocks. We then present a simplified variant that only assigns blocks to partitions.

Two-Tiered Bin Packing: This algorithm begins with the current load on each partition and the list of hot tuples. The integer program has a decision variable for each possible partition-tuple assignment, and the constraints allow each hot tuple set to be assigned to exactly one partition. In addition, there is a decision variable for the partition assignment of each block of B cold tuples. The program calculates each partition's load by summing the access counts of its assigned hot and cold tuple sets. The final constraint specifies that each partition has an equal share of the load, $\pm\varepsilon$. Therefore, if A is the average load over all partitions, the resulting load on partition p_j must be in the range $A-\varepsilon \leq L(p_j) \leq A+\varepsilon$. The planner's objective function minimizes tuple movement while adhering to each partition's capacity constraints, thereby favoring plans with lower network bandwidth requirements.

For each potential assignment of a hot tuple r_i to partition p_j, there is a binary decision variable $x_{i,j} \in \{0,1\}$. Likewise, for each potential assignment of a cold tuple block b_k to partition p_j, there is a variable $y_{k,j} \in \{0,1\}$. In the following equations, we assume a database with n hot tuples, d cold tuple blocks, and c partitions. As defined in Section 3.3.2, $L(r_i)$ is the load (access count) on tuple r_i. Our first constraint requires that each hot tuple set is assigned to exactly one partition, so for each hot tuple set r_i,

$$\sum_{j=1}^{c} x_{i,j} = 1 \tag{3.1}$$

Likewise, for each cold block b_k,

$$\sum_{j=1}^{c} y_{k,j} = 1 \tag{3.2}$$

We seek a balanced load among the partitions, giving them a target load of $A \pm \varepsilon$, so for each partition p_j,

$$L(p_j) = \sum_{i=1}^{n} (x_{i,j} \times L(r_i)) + \sum_{k=1}^{d} (y_{k,j} \times L(b_k)) \geq A - \varepsilon \tag{3.3}$$

If a tuple is not assigned to its original partition according to the reconfiguration plan, it has a transmission cost of T. We assume that all machines in the cluster are located in the same data center, and therefore the transmission cost between any two partitions is the same. Thus, without loss of generality, we can set $T = 1$. We represent the transmission cost of assigning tuple r_i to partition p_j as a variable $t_{i,j} \in \{0, T\}$. Our objective function selects placements with reduced transmission overhead. Hence, it minimizes:

$$\sum_{i=1}^{n} \sum_{j=1}^{c} (x_{i,j} \times t_{i,j}) + \sum_{k=1}^{d} \sum_{j=1}^{c} (y_{k,j} \times t_{k,j} \times B) \tag{3.4}$$

Clearly, moving individual hot sets is less expensive than transmitting blocks of B cold tuples.

One-Tiered Bin Packing: This is the same procedure as the 2-tiered algorithm but without using a list of hot tuples. Hence, rather than dividing the problem into hot and cold parts, all tuples are assigned using a single planning operation in which data is managed in blocks of size B. This scheme saves on monitoring costs as it does not require tuple tracking, but it is not able to generate a fine-grained partitioning scheme. One-tiered bin packing simulates traditional one-level partitioning schemes [76, 20]. This approach may perform well when data access skew is small, but it is unlikely to work in the presence of substantial skew.

3.4.3 Approximate Placement

The bin packing algorithms provide a baseline for optimal reconfiguration of the database, but they are not practical for most applications. Because E-Store is intended for use in OLTP applications where performance is paramount, we set out to design algorithms capable of producing high quality partition plans in a much shorter timeframe – on the order of seconds rather than hours. To this end, we implemented the following practical algorithms to assign hot tuples and cold blocks to partitions.

Greedy: This simple approach assigns hot tuples to partitions incrementally via locally optimal choices. It iterates through the list of hot tuples starting with the most frequently accessed one. If the partition currently holding this tuple has a load exceeding the average $A \pm \varepsilon$ as in Section 3.4.2, the Greedy algorithm sends the tuple to the least burdened partition. It continues to the next most popular tuple until all have been redistributed. Although this algorithm operates in linear time, its usefulness is limited because this scheme only makes locally optimal decisions. It also does not move any blocks of cold tuples, which could impact its performance on workloads with lower levels of skew.

Greedy Extended: This algorithm first executes the Greedy algorithm for hot tuples. If one or more partitions are still overburdened after rebalancing, this scheme executes a similar operation with the cold blocks. Each over-burdened server sends blocks of B tuples to the server with the lowest load until all partitions are within the bounds of capacity. The Greedy Extended algorithm's runtime is comparable to that of the standard Greedy algorithm.

First Fit: This approach globally repartitions the entire database using a heuristic that assigns tuples to partitions one at a time. It begins with the list of hot tuples sorted by their access frequency. The scheme places the hottest tuple at partition 0. It continues to add hot tuples to this partition until it has reached capacity, at which point the algorithm assigns tuples to partition 1. Once all the hot tuples have been placed, the algorithm assigns cold blocks to partitions, starting with the last partition receiving tuples. This approach favors collocating hot tuples and runs in constant time. In some circumstances it leads to better utilization of the DBMS's CPU caches, because hot partitions serve fewer items. But it

also makes the first partitions more vulnerable to overload because they are handling the hottest data. Moreover, because this algorithm does not make any attempt to minimize the movement of tuples during reconfiguration, the migration process may be expensive and cause temporary performance degradation. The evaluation shows that First Fit is inferior to the Greedy Extended approach for the reasons just mentioned, but we include it for completeness as it is a popular bin packing algorithm used in other work [21, 80, 105].

In summary, we propose three techniques for managing data placement for an elastically scaling transaction processing system. We use one- and two-tiered bin-packing "oracles" to construct optimal partitioning plans. We then compare these unrealistic approaches to the more practical approximation algorithms.

3.5 Evaluation

We now present our evaluation of the E-Store framework integrated with H-Store. We conducted an extensive set of experiments using large datasets and three different benchmarks to analyze the parameter sensitivity and performance of E-Store. We report our time-series results using a sliding-window average.

All of the experiments were conducted on a cluster of 10 Linux nodes connected by a 10Gb switch. Each node has two Intel Xeon quad-core processors running at 2.67GHz with 32GB of RAM. We used the latest version of H-Store with command logging enabled to write out transaction commit records to a 7200 RPM disk. We did not find logging to be a significant bottleneck in our experiments.

3.5.1 Benchmarks

We now describe the workloads that we used in our evaluation. For all three benchmarks, we examined three access patterns; no skew, low skew, and high skew.

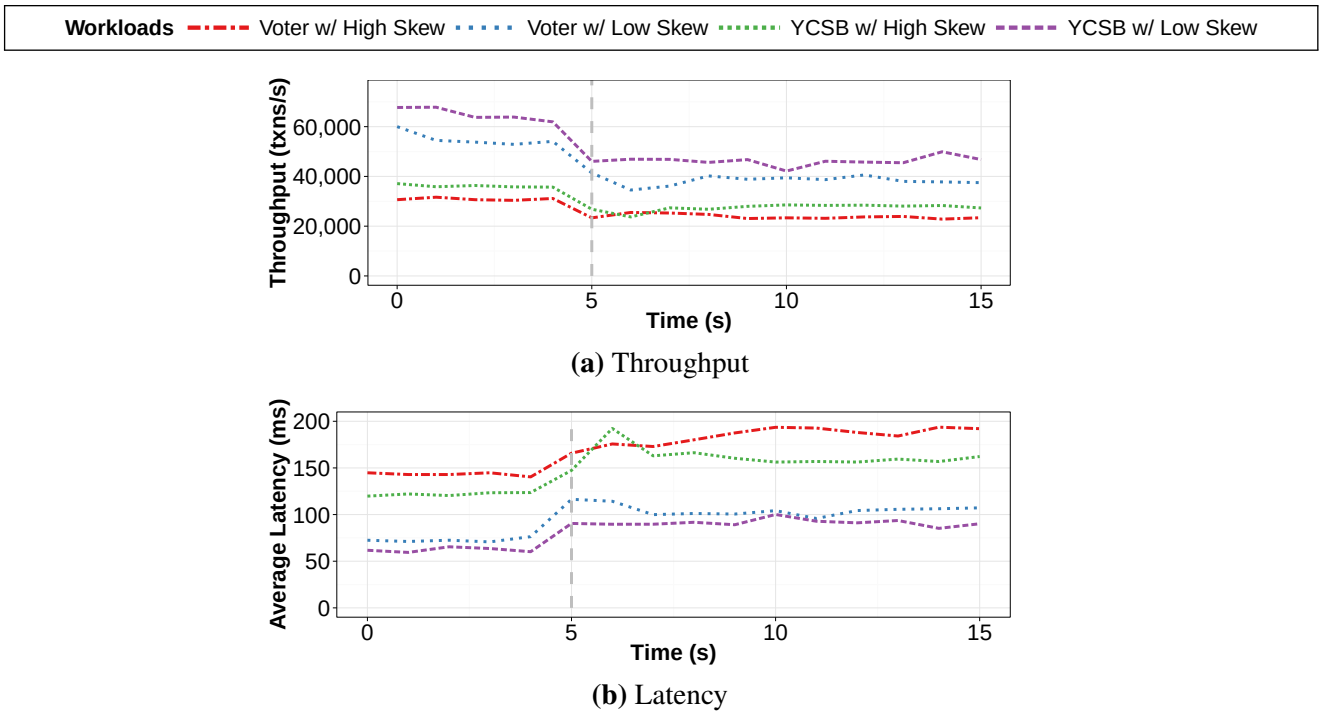
Voter: The Voter Benchmark [90] simulates a phone-based election application. It is designed to saturate the DBMS with many short-lived transactions that all update a small number of records. The database consists of three tables. Two tables are read-only and replicated on all servers: they store the information related to contestants and map area

codes to the corresponding locations. The third table stores the votes and it is partitioned; the telephone number of the voter is used as the partitioning attribute. An individual is only allowed to vote a fixed number of times. As mentioned above, we use three different types of skew: no skew, low skew, and high skew. Low skew simulates local interest in the contest, and is modeled by a Zipfian distribution where two-thirds of the accesses go to one-third of the tuples. High skew simulates highly localized interest where 30 phone numbers are responsible for attempting to vote 80% of the time. The remaining 20% of votes follow the Zipfian distribution described above. The 30 hot phone numbers will use up their allowed votes, but their continued effort to vote will strain database resources on their partition.

YSCB: The Yahoo! Cloud Serving Benchmark has been developed to test key-value data stores [19]. It consists of a single table partitioned on its primary key. In our experiments, we configured the YCSB workload generator to execute 85% read-only transactions and 15% update transactions, with no scans, deletes or inserts. Since Voter is write-heavy, we ran YCSB with a read-bias for balance. We used a database with 60 million tuples that are each 1KB ($\sim$60GB in total). Again we ran no skew, low skew and high skew cases. The no skew case uses a baseline uniform distribution. The low skew case uses a Zipfian distribution where two-thirds of the accesses go to one-third of the tuples. The high skew case uses the same low-skew Zipfian distribution applied to 40% of the accesses, combined with additional "hotspots", where the remaining 60% of the accesses go to 40 individual tuples in partition 0. These definitions are the same as those used in Section 3.1.

TPC-C: This is an industry-standard benchmark for OLTP applications that simulates the operation of a wholesale parts-supply company [96]. The company's operation is centered around a set of warehouses that each stock up to 100,000 different items. Each warehouse has ten districts, and each district serves 3000 customers. The five transactions (and their percentage of the total workload) are:

1. Adding a new customer order (45%)
2. Recording payment from a customer (43%)
3. Making a delivery (4%)



(a) Throughput

(b) Latency

Figure 3-8: The impact of tuple-level monitoring on throughput and latency. Dashed lines at 5 seconds indicate the start of tuple-level monitoring.

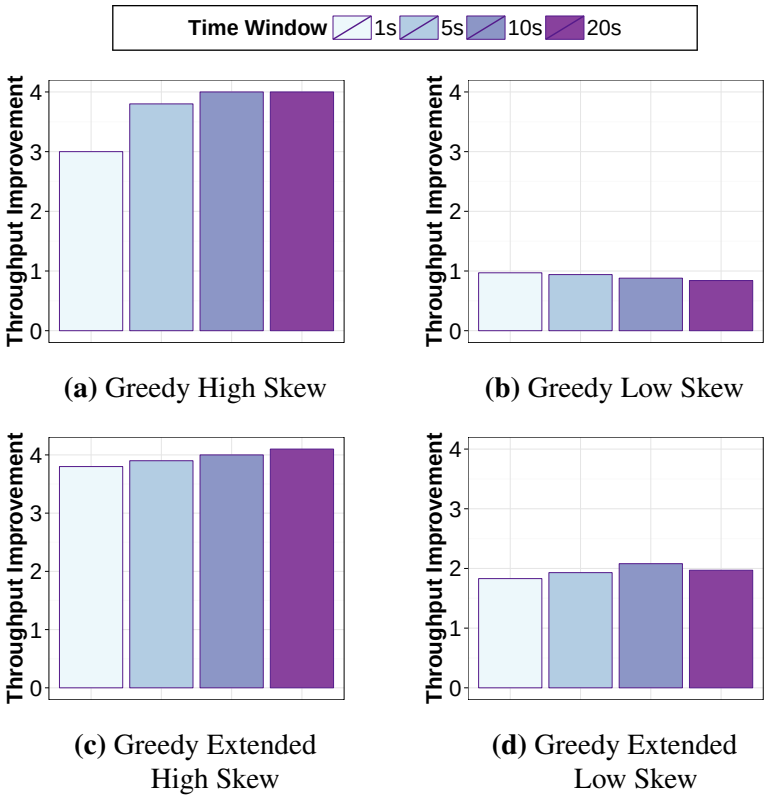
4. Checking the status of an order (4%)

5. Checking the stock level of a warehouse (4%)

For these experiments, we ran TPC-C on a database with 100 warehouses. We again tested three different skew settings. For low-skew, we used a Zipfian distribution where two-thirds of the accesses go to one-third of the warehouses. For the high-skew trials, we modified the distribution such that 40% of accesses follow the Zipfian distribution described above, and the remaining 60% of accesses target three warehouses located initially on partition 0. According to the TPC-C standard, 90% of the time customers can be served by their home warehouse, so if the tables are partitioned by their `WAREHOUSE` id, at most 10% of the transactions will be multi-partitioned [76].

3.5.2 Parameter Sensitivity Analysis

Once E-Store decides that a reconfiguration is needed, it turns on tuple-level monitoring for a short time window to find the top-k list of hot tuples. We analyzed the E-Store per-



(a) Greedy High Skew

(b) Greedy Low Skew

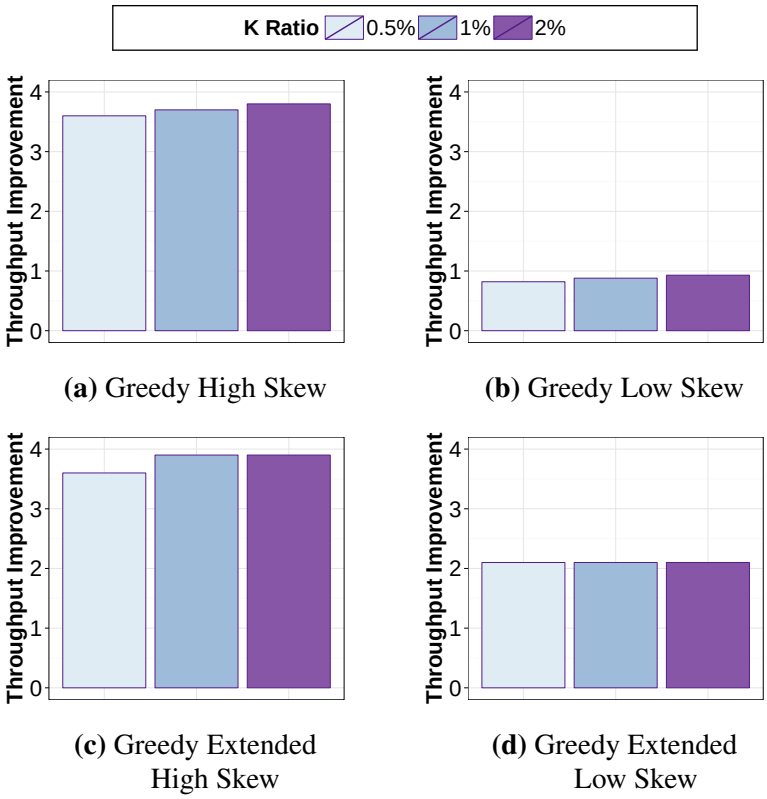
(c) Greedy Extended High Skew

(d) Greedy Extended Low Skew

Figure 3-9: Throughput improvement ratio for YCSB after reconfiguration with Greedy and Greedy Extended planners with different time windows.

formance degradation in terms of throughput and latency due to this tracking. In each trial, we first executed the target workload for 60 seconds to let the system warm-up. We then collected the throughput and latency measurements. After five seconds, we enabled tuple-level monitoring with the top-k percentage of tracked tuples set to 1%. The results in Figure 3-8 show that the monitoring reduces throughput by $\sim$25% for the high skew workload and $\sim$33% in the case of low skew. Moreover, the latency increases by about 45% in the case of low skew and about 28% in the case of high skew. This performance degradation is due to the way tuple accesses are counted in H-Store: accesses to memory locations in the C++-based execution engine must be mapped to their corresponding tuple ID, which requires a reverse index lookup with a Java Native Interface call that adds significant overhead to each transaction. In future work we plan to investigate ways to reduce this overhead.

We next analyzed the sensitivity of the monitoring time window W and top-k ratio parameters. Figure 3-9 shows the throughput improvement ratio (throughput after reconfigu-



(a) Greedy High Skew

(b) Greedy Low Skew

(c) Greedy Extended High Skew

(d) Greedy Extended Low Skew

Figure 3-10: Throughput improvement ratio for YCSB after reconfiguration with Greedy and Greedy Extended planners with different top-k ratios.

ration divided by throughput before reconfiguration) for the Greedy and Greedy Extended planners with time windows of variable length. The figure shows that the Greedy Extended algorithm is not sensitive to variation in the length of the time window. In contrast, the Greedy algorithm shows some sensitivity to the length of the time window since it is more dependent on the accuracy of the detected hot tuples set. Note that our measure of throughput after reconfiguration includes the monitoring and reconfiguration periods during which throughput is reduced, so a longer monitoring interval sometimes results in lower performance.

Lastly, we conducted an experiment for the top-k ratio, for $k = 0.5\%$, 1%, and 2%. Figure 3-10 illustrates that both Greedy and Greedy Extended algorithms are not sensitive to variation in this parameter. This is probably due to the fact that in our experiments we had a small number of hot tuples and a large number of cold tuples. Therefore it is likely that nearly all of the hot tuples were found among the top-k, even for $k = 0.5\%$. We would probably see more sensitivity with a much larger number of hot tuples (e.g., a

Planner	Low skew	High skew
One-tier bin packer	$>$ 20 hrs	$>$ 20 hrs
Two-tier bin packer	$>$ 20 hrs	$>$ 20 hrs
Greedy	835 ms	103 ms
Greedy Extended	872 ms	88 ms
First Fit	861 ms	104 ms

Table 3.1: Execution time of all planner algorithms on YCSB.

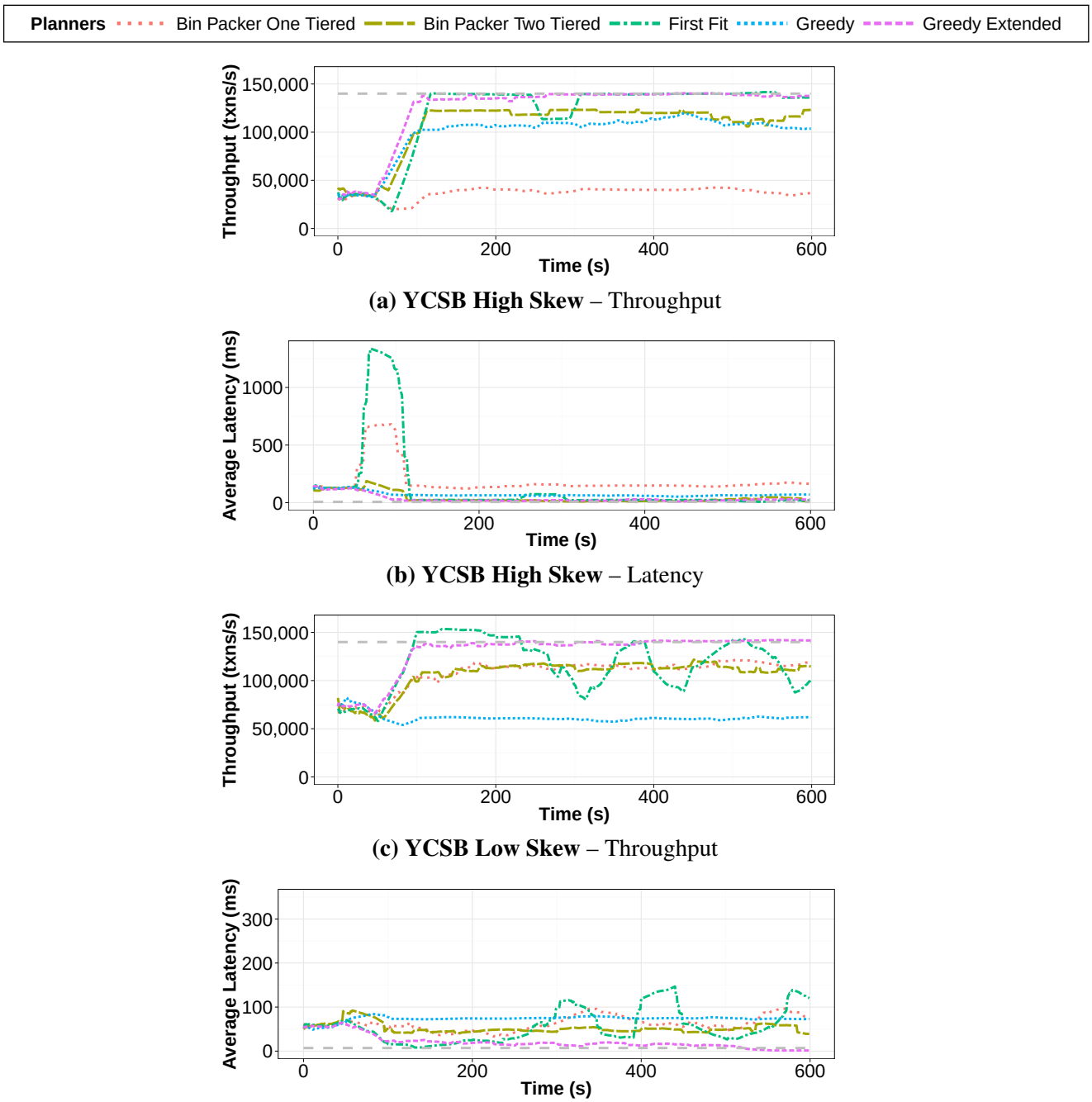
pathological case in which every tuple in partition 0 is hot) or a much smaller top-k ratio. These experiments show that E-Store is robust to parameter settings under most normal use cases, and it is safe to use the default values under different levels of skew. As such, we use a time window of 10 seconds and top-k ratio of 1% for all the remaining experiments in this chapter.

3.5.3 One-Tiered vs. Two-Tiered Partitioning

We next compared the efficacy of the plans generated by the one- and two-tiered placement algorithms for load balancing (with no servers added or removed). For this experiment, we used the YCSB workload with low and high skew. We implemented both of the bin packing algorithms from Section 3.4.2 inside of the E-Planner using the GLPK solver[1]. Since these algorithms find the optimal placement of tuples, this experiment compares the ideal scenario for the two different partitioning strategies. The database's tuples are initially partitioned uniformly across five nodes in evenly sized chunks. E-Store moves tuples among these five nodes to correct for the load imbalance. E-Monitor and E-Planner run as standalone processes on a separate node.

Figures 3-11 and 3-12 show the results of running the two bin packer algorithms (and others to be discussed in the next section) on the various types of skew to balance load across the five nodes for the YCSB workload. Note that the time to compute the optimal plan is exceedingly long for an on-line reconfiguration system like E-Store (see Table 3.1). Thus for these experiments, we terminated the solver after 20 hours; we did not observe a noticeable improvement in the quality of the plan beyond this point.

[1] `http://www.gnu.org/s/glpk/`



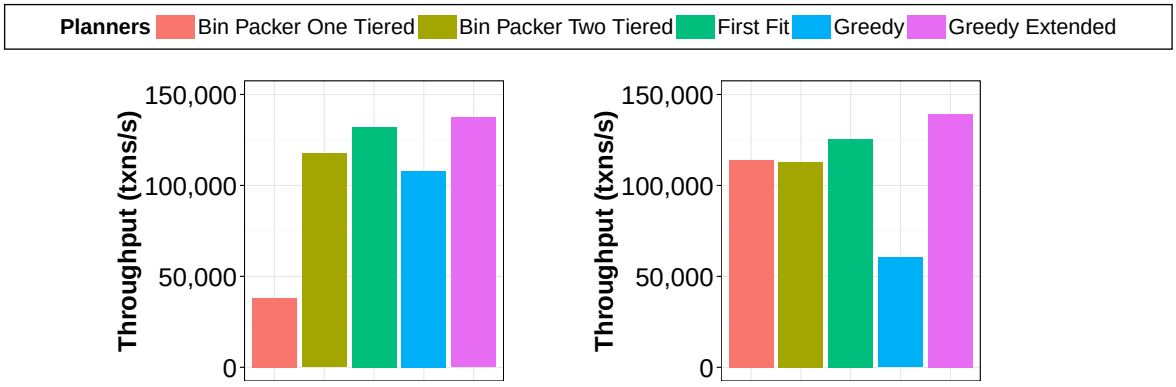
(a) YCSB High Skew – Throughput

(b) YCSB High Skew – Latency

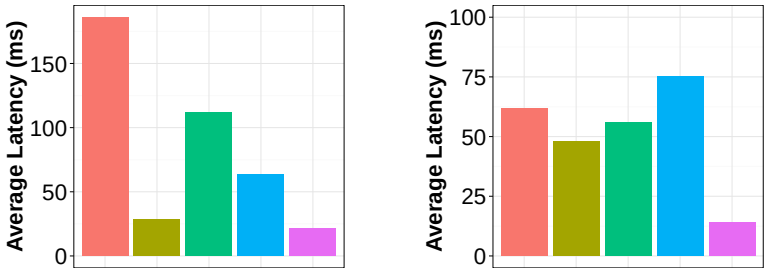
(c) YCSB Low Skew – Throughput

(d) YCSB Low Skew – Latency

Figure 3-11: Comparison of all our tuple placement methods with different types of skew on YCSB. In each case, we started E-Store 30 seconds after the beginning of each plot. Since we are only concerned with load-balancing performance here, we skipped phase 1 of E-Monitor. The dashed gray line indicates system performance with no skew.



(a) YCSB High Skew – Throughput **(b) YCSB Low Skew** – Throughput



(c) YCSB High Skew – Latency **(d) YCSB Low Skew** – Latency

Figure 3-12: YCSB throughput and latency from Figure 3-11 averaged from the start of reconfiguration at 30 seconds to the end of the run.

The reason the optimal plan takes so long to compute is that there are a large number of inputs to the integer linear program. For the high skew YCSB experiment with the two-tiered bin packer, there are 2003 rows, 59190 columns, and 118380 non-zeros in the input matrix representing the binary decision variables and constraints of the problem (see Section 3.4.2). A recent paper using a similar linear program formulation for allocating data to servers found that the optimal allocation could only be found with a maximum of seven "backends" [78]. In this experiment we used thirty partitions, so it is no surprise that the program did not finish in a reasonable amount of time.

In Figure 3-11 and all subsequent performance vs. time plots, tuple-level monitoring starts 30-seconds after the beginning of the plot. The 20 hours to compute the placement plan for the one- and two-tiered bin packer algorithms is not shown in the plots, for obvious reasons. The horizontal dashed gray line indicates system performance with no skew (a uniform load distribution). The goal of E-store is to achieve the same level of performance as the no-skew case even in the presence of skew. The drop in throughput and increase in latency around 30 seconds is due to the overhead of reconfiguration.

Both optimal bin packer algorithms perform comparably well in the case of low skew, however the DBMS achieves a lower latency more quickly with the two-tiered approach. Moreover, the two-tiered approach performs better in the high skew workload since it identifies hot spots at the individual tuple level and balances load by redistributing those tuples. The two-tiered approach is able to balance load such that throughput is almost the same as the no skew workload.

3.5.4 Approximate Placement Evaluation

The main challenge for our approximate placement algorithms is to generate a reconfiguration plan in a reasonable time that allows the DBMS to perform as well as it does using a plan generated from the optimal algorithms. For these next experiments, we tested our three approximation algorithms from Section 3.4.3 on YCSB and Voter workloads with both low and high skew. All tuples are initially partitioned uniformly across five nodes. Then during each trial, E-Store moves both hot tuples and cold blocks between nodes to

correct for load imbalance caused by skew.

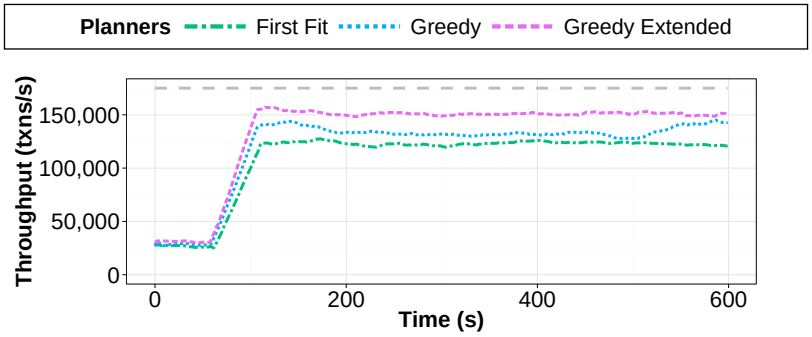
Figures 3-11 and 3-12 show the DBMS's performance using E-Store's approximate planners for the two different skew levels for YCSB. These results are consistent with our results with the Voter workload reported in Figures 3-13 and 3-14.

In the case of high skew, all three approximate planners perform reasonably well, but Greedy Extended and Greedy stabilize more quickly since they move fewer tuples than First Fit. After stabilizing, Greedy Extended and First Fit both perform comparably to the two-tiered bin packer approach. Specifically, Figure 3-11a shows a $4\times$ improvement in throughput and Figure 3-11b shows a corresponding $10\times$ improvement in latency. Greedy Extended performs the best overall, since it avoids the spike in latency that First Fit exhibits as a result of moving a large number of tuples during reconfiguration.

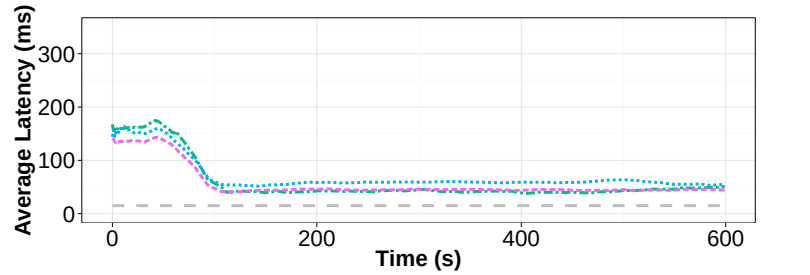
In the case of low skew, Greedy Extended also produces the best reconfiguration plan since it reaches a stable throughput and latency that is better than the others more quickly. The plan generated by First Fit achieves good performance too, but it does not stabilize within the 10 minute window since it moves such a large amount of data. Reconfiguration of large chunks of data takes time because Squall staggers the movement of data to avoid overloading the system (recall Section 3.2.1).

In summary, Greedy Extended produces the same performance as the two-tiered bin packer approach and runs in just a few seconds. We note that because the Greedy algorithm only considers hot tuples, it does not generate good plans for workloads with low skew. This provides additional evidence of the importance of considering both hot tuples and cold blocks.

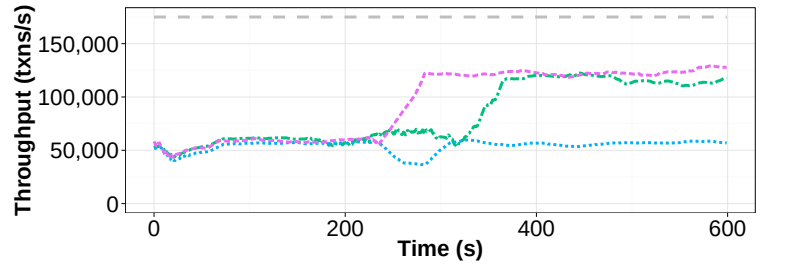
To gauge the effectiveness of E-Store on applications with few root nodes in the tree schema, we also ran two experiments with TPC-C. In our TPC-C experiments there are only 100 root tuples and all the other tuples are co-located with these ones. Hence, our Greedy Extended scheme is overkill and it is sufficient to use the Greedy allocation scheme, which only looks at hot tuples. In the TPC-C experiments, the 100 warehouses were initially partitioned across three machines in evenly sized chunks, with skew settings as described in Section 3.5.1. As shown in Figure 3-15, E-Store improves both the latency and throughput of TPC-C under the two different levels of skew. The impact of reconfiguration is larger



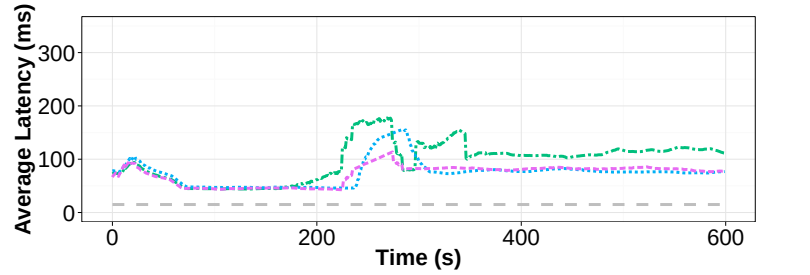
(a) Voter High Skew – Throughput



(b) Voter High Skew – Latency

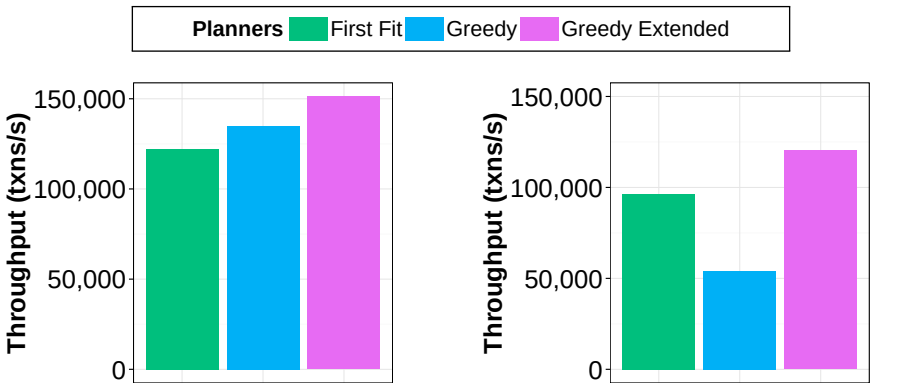


(c) Voter Low Skew – Throughput



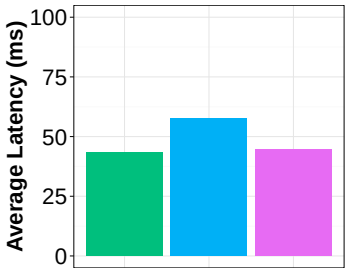
(d) Voter Low Skew – Latency

Figure 3-13: Comparison of approximate tuple placement methods with different types of skew on Voter. The dashed gray line indicates system performance with no skew.

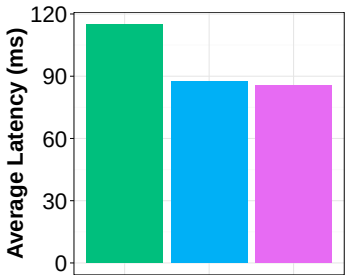


(a) Voter High Skew – Throughput

(b) Voter Low Skew – Throughput

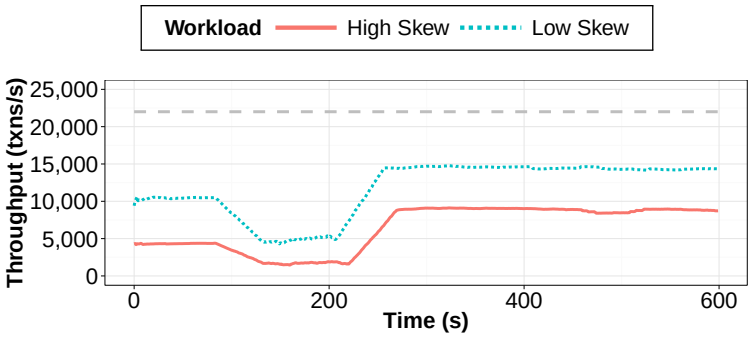


(c) Voter High Skew – Latency

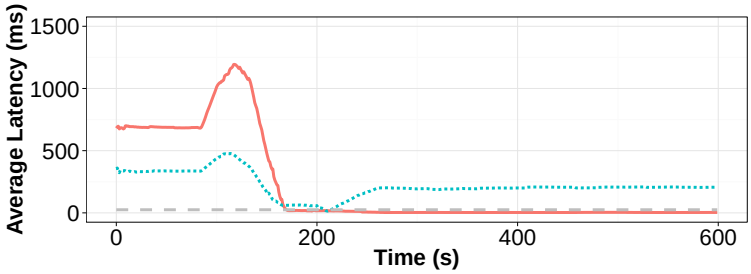


(d) Voter Low Skew – Latency

Figure 3-14: Voter throughput and latency from Figure 3-13, averaged from the start of reconfiguration at 30 seconds to the end of the run.



(a) Throughput



(b) Latency

Figure 3-15: The Greedy planner with different types of skew on a TPC-C workload. The dashed gray line indicates system performance with no skew (a uniform load distribution).

for TPC-C than the other benchmarks for a few reasons. First, each warehouse id has a significant amount of data and tuples associated with it. Therefore, reconfiguring TPC-C requires more time and resources not only to move all data associated with each warehouse, but also to extract and load a large number of indexed tuples. Second, as roughly 10% of transactions in TPC-C are distributed, a migrating warehouse can impact transactions on partitions not currently involved in a migration. For these reasons, load-balancing TPC-C can require longer to complete, but it results in a significant improvement in both throughput and latency.

3.5.5 Performance after Scaling In/Out

We next measured E-Store's ability to react to load imbalance by increasing and decreasing the DBMS's cluster size. We tested both overloading and underloading the system with the two different levels of skew to prompt E-Store to scale out or in. We used the YCSB and Voter workloads again with tuples initially partitioned across five nodes in evenly sized blocks. We only evaluated plans using the Greedy Extended algorithm, since our previous

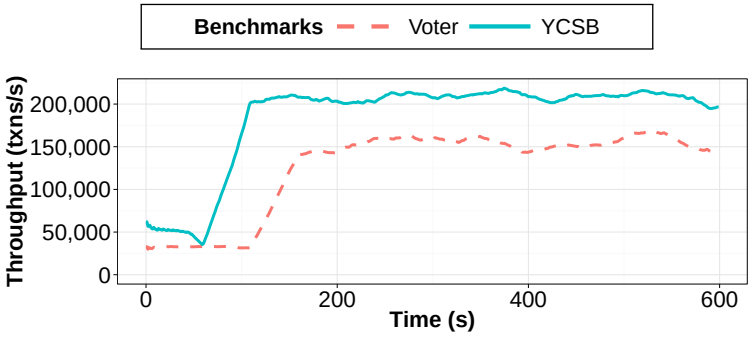
experiments demonstrated its superiority for these workloads.

E-Store can scale out with minimal overhead in order to handle a system that is simultaneously skewed and overloaded. Figure 3-16 shows the results of overloading the system and allowing E-Store to expand from five to six nodes. We also tested E-Store's ability to remove nodes when resources are underutilized. Figure 3-17 shows the results of underloading the system and allowing E-Store to scale in from five to four nodes. These experiments show that E-Store maintains system performance when scaling in, other than a brief increase in latency due to migration overhead. In the case of high skew, E-Store actually improves performance due to load balancing, despite using fewer nodes and, hence, fewer partitions.

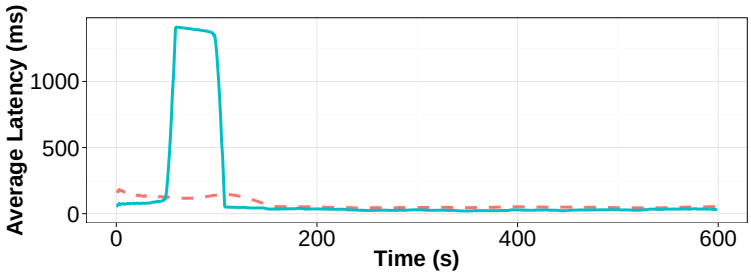
The next chapter focuses more on the problem of scaling in and out, and introduces the second system in this thesis, P-Store. The experiments will show that E-Store is capable of reactively scaling out from one to seven machines by adding one machine at a time. For such large load increases, however, E-Store incurs large latency spikes during each reconfiguration. The next chapter will show that for large, predictable increases in aggregate load it is better to scale proactively as P-Store does, by adding one or more machines at a time in advance of load increases.

3.6 Conclusion

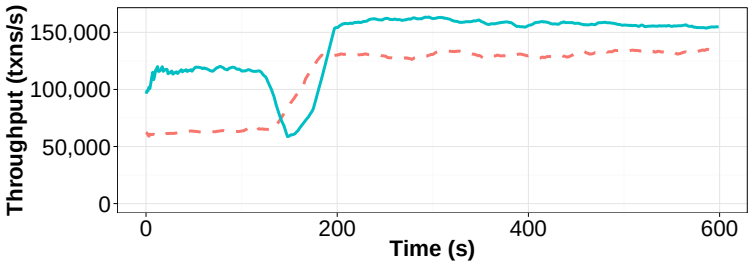
This chapter has described the E-Store system for database elasticity. E-Store manages skew using a novel two-tiered partitioning scheme. In this scheme, a small number of frequently accessed, "hot" tuples are partitioned explicitly, and the rest of the "cold" tuples are partitioned in large blocks. E-Store's end-to-end framework satisfies the requirements for an ideal elastic system as described in Chapter 1. It provides transactional ACID guarantees by integrating with an OLTP DBMS such as H-Store. It ensures that no manual intervention is required by continually monitoring the system for imbalances in CPU utilization, and automatically triggers an elastic reconfiguration if an imbalance is detected. Once elastic reconfiguration is triggered, a tuple-level monitoring component called *E-Monitor* is enabled for a short period of time to identify hot tuples. Next, a planning component



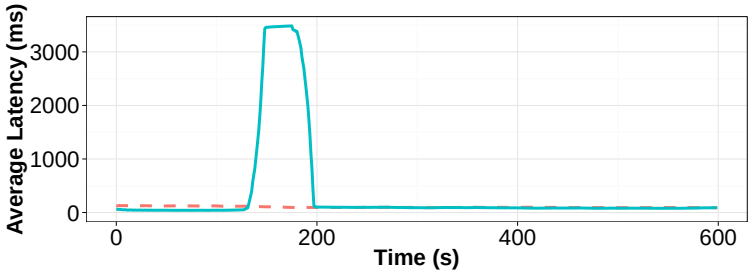
(a) High Skew – Throughput



(b) High Skew – Latency

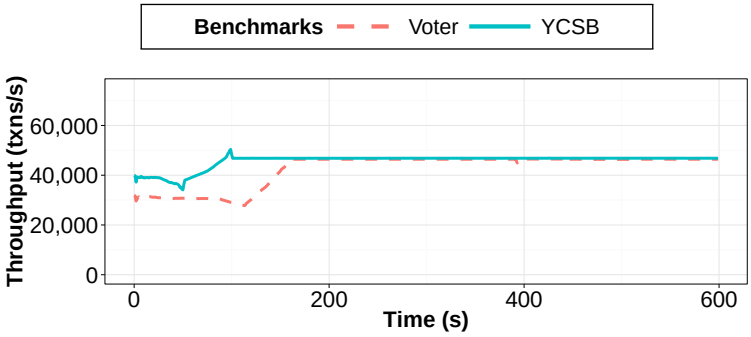


(c) Low Skew – Throughput

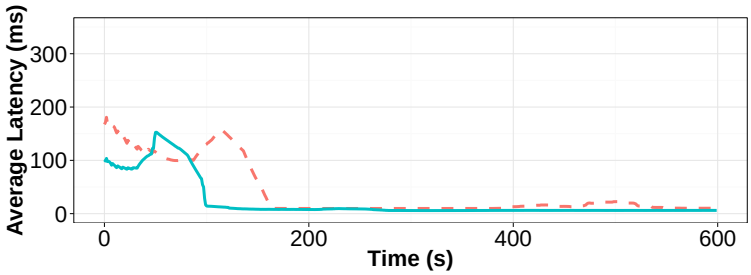


(d) Low Skew – Latency

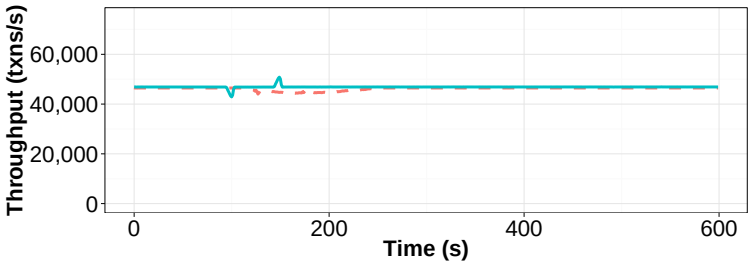
Figure 3-16: The Greedy Extended planner with different types of skew on Voter and YCSB workloads. In these experiments we overloaded the system, causing it to scale out from 5 to 6 nodes.



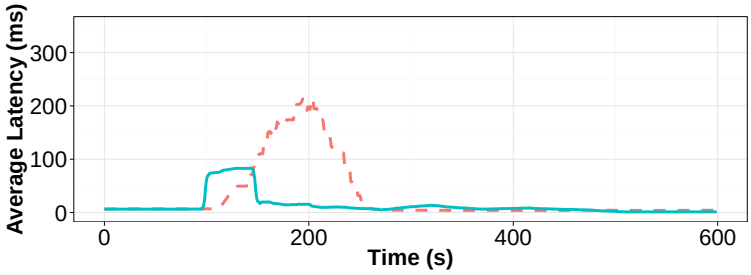
(a) High Skew – Throughput



(b) High Skew – Latency



(c) Low Skew – Throughput



(d) Low Skew – Latency

Figure 3-17: The Greedy Extended planner with different types of skew on Voter and YCSB workloads. In these experiments we underloaded the system, causing it to scale in from 5 to 4 nodes.

called *E-Planner* determines how to re-partition the database in order to balance the workload, using the two-tiered partitioning scheme. Finally, a live migration component called *Squall* physically moves data so that the data layout matches the new plan from E-Planner. The evaluation shows that the E-Store framework can effectively handle a variety of workloads and types of skew, increasing throughput by up to $4\times$ while reducing latency by up to $10\times$.

Chapter 4

P-Store

Many OLTP workloads follow a predictable, diurnal pattern in which load during the day can be an order of magnitude larger than load at night (recall Figure 1-1). Although a reactive system such as E-Store will correctly scale in and out for these workloads as the load varies throughout the day, it is not ideal; by definition, a reactive system is always one step behind. This delayed response can be a problem if it is necessary to move a large amount of data during each reconfiguration, since performance may be degraded for an extended period of time. E-Store quickly corrects load imbalances due to high skew by moving a small amount of data, but if skew is low and the database is large, corrective action may take much longer.

This chapter describes P-Store, an elastic system which uses predictive modeling to proactively scale out in advance of load increases. P-Store determines how many servers will be required in the future based on predictions of future load, and calculates how much data will need to move. Based on the amount of data moving, P-Store can determine how long reconfiguration will take, and therefore how far in advance the reconfiguration must start in order to complete in time for the predicted load increase. This chapter will describe in detail how P-Store performs these calculations using a state-of-the-art time series prediction model, a novel dynamic programming algorithm, and a finely tuned model of reconfiguration. The chapter concludes with an evaluation of both E-Store and P-Store on the real database workload of B2W Digital (B2W), the largest online retailer in South America.

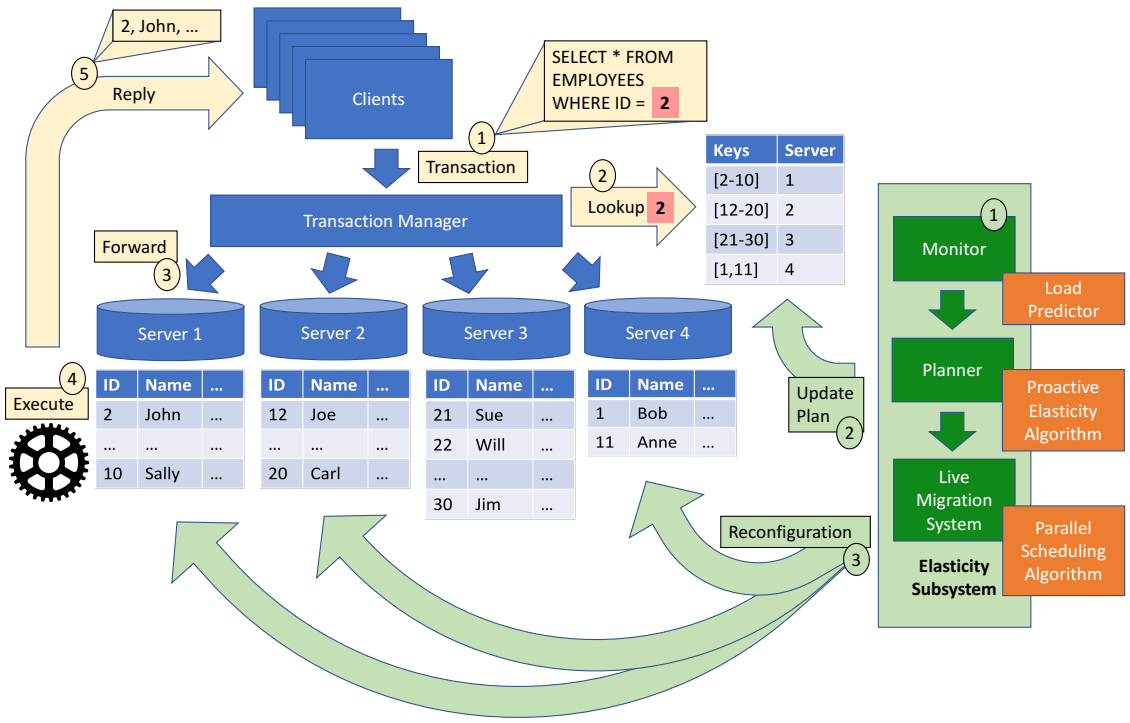
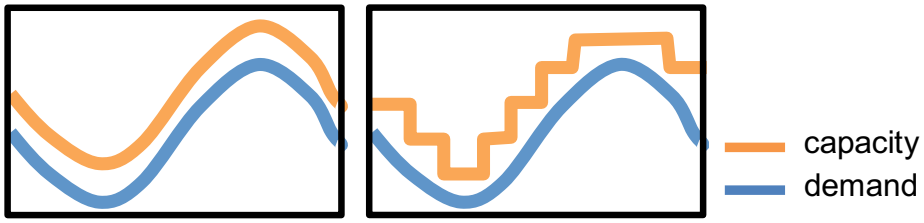


Figure 4-1: Schematic of a shared nothing, partitioned DBMS with an elasticity subsystem. Special components of P-Store are shown in the orange boxes.

Figure 4-1 shows P-Store's major extensions to the model introduced in Chapter 2. As shown in the orange boxes, P-Store deviates from the basic elasticity model by including a *load predictor* to predict future aggregate load on the DBMS given historical data collected by the Monitor. The Planner feeds these predictions into its *proactive elasticity algorithm* to determine when and how to reconfigure the database. At reconfiguration time, a *parallel scheduling algorithm* dictates the schedule for data migration.

4.1 Problem Statement

We now define the problem that predictive elasticity seeks to solve. We consider a DBMS having a *latency constraint*. The latency constraint specifies a service level agreement for the system: for example, that 99% of the transactions must complete in under 500 milliseconds. The *predictive elasticity problem* we consider in this chapter entails minimizing the cost C over T time intervals:



(a) Ideal Capacity **(b)** Actual Servers Allocated

Figure 4-2: Ideal capacity and actual servers allocated to handle a sinusoidal demand curve

$$C = \sum_{t=1}^{T} s_t \tag{4.1}$$

where s_t is the number of servers in the database cluster at time t. For convenience, we have included these and other symbols used throughout the chapter in a table in Appendix A.

Note that this is a different problem than a reactive system such as E-Store seeks to solve. E-Store is only triggered when there is an existing performance problem or the system is underutilized. E-Store's goal is to fix performance issues as quickly as possible while keeping the system available, so it will likely cause violations to the latency constraint until reconfiguration is completed. Conversely, P-Store's goal is to minimize cost while preventing latency violations from happening at all.

Given a constant workload, it is relatively easy to solve this optimization problem: choose a constant number of servers s_t for all t that minimizes C subject to the latency constraint; use techniques from E-Store to handle data skew. The problem becomes more interesting with a variable workload, because we must decide when to reconfigure the database (if at all), and which data to move between servers.

A solution to the predictive elasticity problem must indicate *when* to initiate each reconfiguration and the *target* number of servers for each reconfiguration. In order to minimize the cost in Equation (4.1) and respect the latency constraint, our system should try to make the database's capacity to handle queries as close as possible to the demand while still exceeding it. In the ideal case, the capacity curve would exactly mirror the demand curve with a small amount of buffer (see Figure 4-2a). In reality, we can only have an integral number of servers at any given time, so the actual number of servers allocated must follow a step function (see Figure 4-2b). This must be taken into consideration when minimizing

the gap between the demand function and the capacity curve.

An additional complexity in the predictive elasticity problem is that this step function is actually an approximation of the capacity of the system. The *effective capacity* of the system does not change immediately after a new server is added; it changes gradually as data from the existing servers is offloaded onto the new server, allowing this new machine to help serve queries.

The next section describes how P-Store manages this complexity and solves the predictive elasticity problem.

4.2 Algorithm for Predictive Elasticity

This section describes P-Store's algorithm for predictive elasticity. First, we describe the preliminary offline analysis that needs to be performed on the DBMS to extract key parameters, such as the capacity of each server (Section 4.2.1). Then we introduce the key assumptions and discuss the applicability of the algorithm (Section 4.2.2). Next, we introduce P-Store's algorithm to solve the predictive elasticity problem. The algorithm determines a sequence of reconfigurations that minimizes cost and respects the latency constraint of the application (Section 4.2.3). Finally, we show how the timing and choice of reconfigurations depend on the way reconfigurations are scheduled (Section 4.2.4).

The other important component of P-Store, beyond the predictive elasticity algorithm, is the load prediction component, which we will describe in Section 4.3. Finally, Section 4.4 describes how we put all these components together to build P-Store.

4.2.1 Parameters of the Model

Our model has three parameters that must be empirically determined for a given workload running on a given database configuration:

1. Q: Target average throughput of each server. Used to determine the number of servers required to serve the predicted load.
2. $\hat{Q}$: Maximum throughput of each server. If the load exceeds this threshold, the latency constraint may be violated.

3. D: Shortest time to move all of the data in the database exactly once with a single sender-receiver thread pair, such that reconfiguration has no noticeable impact on query latency. Reconfigurations scheduled by P-Store will actually move a *subset* of the database with *parallel* threads, but D is used as a parameter to calculate how long a reconfiguration will take so it can be scheduled to complete in time for a predicted load increase. We assume that D increases linearly with database size.

All of these parameters can be determined through offline system evaluation based on the latency constraint of the system as defined in Section 4.1.

Q and $\hat{Q}$ can be empirically determined by running representative transactions from the given workload on a single server, and steadily increasing the transaction rate over time. At some point, the system is saturated and the latency constraint is violated. We set $\hat{Q}$ to 80% of this saturation point to ensure some "slack". Q should be set to some value below $\hat{Q}$ so that normal workload variability does not cause a server's load to exceed $\hat{Q}$. We set Q to 65% of the saturation point in our evaluation.

D is determined by fixing the transaction rate per node at $\hat{Q}$ and executing a series of reconfigurations, where in each reconfiguration we increase the rate at which data is moved. At some point, the reconfiguration starts to impact the performance of the underlying workload and to lead to violations of the latency constraint because there aren't enough CPU cycles to manage the overhead of reconfiguration and also execute transactions. D is set to the reconfiguration time of moving the entire database at the highest rate for which reconfiguration has no noticeable impact on query latency, plus a buffer of 10%. The buffer is needed because D will actually be used to calculate the time to move subsets of the database (not the entire thing), and the data distribution may not be perfectly uniform.

4.2.2 Applicability

The proactive reconfiguration algorithm we will describe in Section 4.2.3 relies on several assumptions, but we believe it is still widely applicable to many OLTP applications. The key assumptions are:

- **Load predictions are accurate to within a small error.** Section 4.3 shows how SPAR, the default predictive model used by P-Store, works well for the common case of diurnal workloads. But our algorithm can be combined with any predictive model if it is well suited for a given workload.
- **The workload mix is not rapidly changing.** This is a reasonable assumption for most OLTP workloads, in which the set of transactions and their distribution do not change very often. When they do change, we can simply measure Q and $\hat{Q}$ again.
- **The database size is not rapidly changing.** This is true of many OLTP applications that keep only active data in the database. Historical data is moved to a separate data warehouse. Any significant size increase or decrease requires re-discovering D.
- **The workload is distributed uniformly across the data partitions.** Different tuples in a database often have different access frequencies, but these differences are smoothed out when the tuples are randomly grouped into data partitions with a good hash function. As a result, the load skew among data partitions is generally much lower than the load skew among tuples. Our evaluation shows that this uniformity assumption is a good approximation for the workload we considered. (If a workload has a tuple that is so hot that a single partition cannot handle it alone, then a partitioned database is probably the wrong choice for that workload.)
- **The data is distributed uniformly across the data partitions.** Similar to the previous point, some keys may have more data associated with them than others, but differences are generally smoothed out at the partition level.
- **The workload has few distributed transactions.** This is an assumption made by many partitioned database systems including H-Store, and is required for the system to scale (almost) linearly.

Although P-Store is implemented in a main memory setting, the ideas should be applicable to most partitioned OLTP DBMSs with some parameter changes.

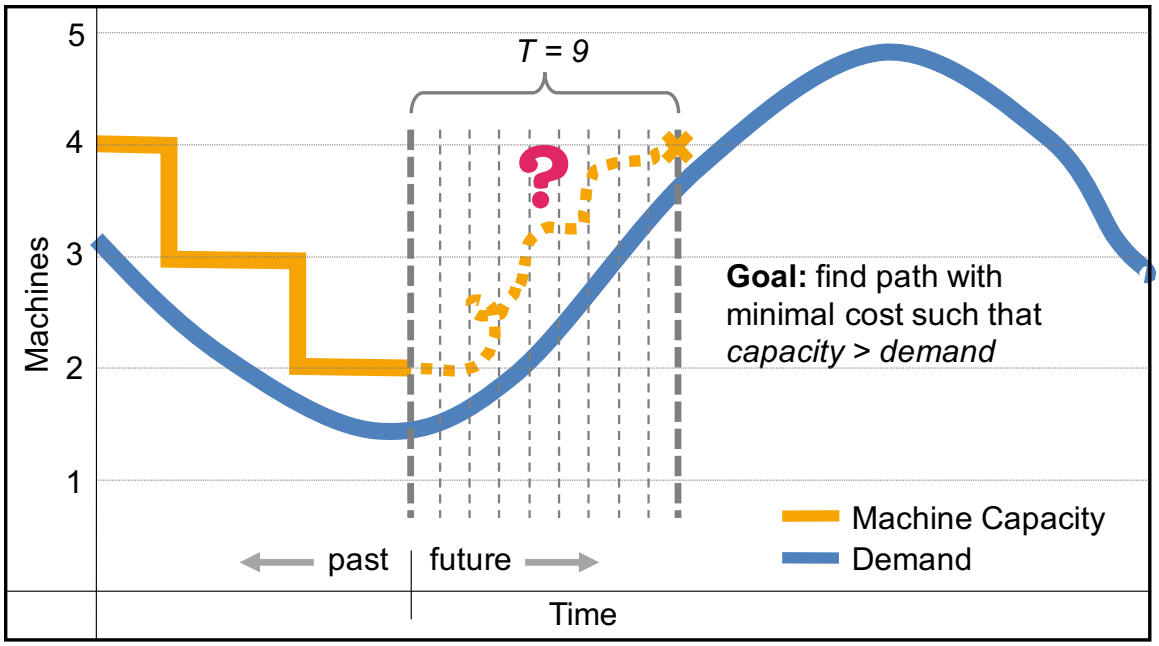


Figure 4-3: Schematic of the goal of the Predictive Elasticity Algorithm.

4.2.3 Predictive Elasticity Algorithm

Our algorithm for proactive reconfiguration must determine when to reconfigure and how many machines to add or remove each time. This corresponds to finding a series of *moves*, where each move consists of adding or removing zero or more machines ("doing nothing" is a valid move).

Formally speaking, a move is a reconfiguration going from B machines before to A machines after (adding $A-B$ machines on scale-out, removing $B-A$ machines on scale-in, or doing nothing if $A=B$). Each move has a specified starting and ending time. We will use the variables B and A and the notion of *move* repeatedly throughout the chapter.

At a high level, our algorithm tries to plan a series of moves spanning a period of time from the present moment to a specified time in the future. For simplicity, we discretize that period of time into T time intervals. Each move therefore lasts some positive number of time intervals (rounded up to the nearest integer). Figure 4-3 shows a schematic of the high level goal of the algorithm. In this schematic, $T=9$ time intervals, and the goal is to find a series of moves starting at $B=2$ machines at $t=0$ and ending at $A=4$ machines at $t=9$, such that capacity exceeds demand and cost is minimized.

Algorithm 1: Calculate the best series of moves for a given time series array of predicted load L and starting number of nodes N_0

```
Function best-moves(L, N_0, Q)
    Input: Time series array of predicted load L of length T, machines allocated
           initially N_0, target average transaction rate per node Q
    Output: Best series of moves M
    // Calculate the maximum number of machines ever needed to serve
    // the predicted load
    Z ← max(⌈max(L)/Q⌉, N_0);
    for i ← 1 to Z do
        // Initialize matrix m to memoize cost and best series of
        // moves
        m ← ∅;
        if cost(T, i, L, N_0, Z, m) ≠ ∞ then
            t ← T; N ← i;
            while t > 0 do
                Add (t,N) to M;
                t ← m[t,N].prev_time;
                N ← m[t,N].prev_nodes;
            Reverse M;
            return M;
    // No feasible solution
    return ∅;
```

The algorithm has three functions: *best-moves*, *cost* and *sub-cost*. *best-moves* is the top-level function which makes calls to *cost*. *cost* and *sub-cost* recursively call each other. The *cost* function starts from the end of the time period with a specified number of servers. It works backwards with recursive calls to *sub-cost* to find a series of moves to get from the initial state to the specified end state with minimal cost (i.e. using the fewest servers on average). *cost* checks that there is sufficient capacity for the predicted load at the beginning and end of every move, and *sub-cost* checks that there is sufficient capacity during each move. *best-moves* is the top-level function which calls *cost* for all the possible end states, and chooses the solution with minimal cost that has sufficient capacity throughout. We now describe each of these functions in more detail.

Top-Level Algorithm

The *best-moves* function in Algorithm 1 is the top-level algorithm to find the optimal sequence of moves. It receives as input a time series array of predicted load L of length T (generated by P-Store's online predictor component, the topic of Section 4.3), as well as N_0, the number of machines currently allocated at time $t = 0$, and the target average transaction rate per node Q from Section 4.2.1. The output of the algorithm is an optimal sequence M of contiguous, non-overlapping moves ordered by starting time.

Algorithm 1 first calculates Z, the number of machines needed to serve the maximum predicted load in L. After calculating Z, Algorithm 1 iteratively tries to find the optimal series of moves ending with i machines, starting with $i = 1$ and incrementing by one each time, with a maximum of Z. It does so by calling the *cost* function, which returns the minimum cost of a feasible sequence of moves ending with i servers at time T (the internals of the *cost* function will be discussed in the next section). A sequence of moves is "feasible" if no server will be overloaded according to the load prediction L. If no feasible sequence exists, the *cost* function returns an infinite cost. Otherwise, the function populates a matrix m of size $T \times Z$ with the optimal moves it has found: $m[t,A]$ contains the last optimal move that results in having A servers at time t. The element $m[t,A]$ contains the time when the move starts, the initial number of servers for the move, and the cost during the move, i.e., the average number of servers used multiplied by the elapsed time.

As soon as Algorithm 1 finds a series of moves that is feasible (i.e., with finite cost), it works backwards through the matrix m to find the memoized series of moves. Then it reverses the list of moves so they correspond to moving forward through time, and it returns the reversed list. It is not necessary to continue with the loop after a feasible solution is found, because all later solutions will end up with a larger number of machines and therefore have a higher cost.

If no feasible solution is found, that means the initial number of machines N_0 is too low, and it is not possible to scale out fast enough to handle the predicted load. This can happen if, for example, there is a news event causing a flash crowd of customers on the site. There are a couple of options in this case: (1) move data quickly to meet capacity demands

by moving larger chunks at a time (which will incur some latency overhead due to data migration), or (2) continue to move data at the regular rate and suffer some overhead due to insufficient capacity. By default, P-Store reacts to unexpected load spikes with option (2). The evaluation in Section 4.6.2 shows the performance of these two strategies.

Finding an Optimal Sequence of Moves

To minimize the cost of the system over time T, we must find a series of moves spanning time T such that the predicted load never exceeds the effective capacity of the system, and the sum of the costs of the moves is minimized. In order to plan a move from B to A machines, we need to determine how long the move will take. The function $T(B,A)$ expresses this time, which depends on the specific reconfiguration strategy used by the system. We will discuss how to calculate $T(B,A)$ in Section 4.2.4. We also need to find out the moves that minimize cost. The cost of a move is computed by the function $C(B,A)$, which will also be described in Section 4.2.4.

In order to determine the optimal series of moves, we have formulated the problem as a dynamic program. This problem is a good candidate for dynamic programming because it carries optimal substructure. The minimum cost of a series of moves ending with A machines at time t is equal to the minimum cost of a series of moves ending with B machines at time $t - T(B,A)$, plus the (minimal) cost of the last optimal move, $C(B,A)$.

This formulation is made precise in Algorithms 2 and 3. Algorithm 2 finds the cost of the optimal series of moves ending at a given time t and number of machines A. The first line of Algorithm 2 checks the constraints of the problem, in particular that t must not be negative, and if $t = 0$ the number of machines must correspond to our initial state, N_0. It also checks that the predicted load at time t does not exceed the capacity of A machines. We assume that the cost of latency violations (see Section 4.1) is extremely high, so for simplicity, we define the cost of insufficient capacity to be infinite. Moving forward through the algorithm, recall that the matrix element $m[t,A]$ stores the last optimal move found by a call to *cost*. But its secondary purpose is for "memoization" to prevent redundant computation. Accordingly, Algorithm 2 checks to see if the optimal set of moves for this configuration has already been saved in m, and if so, it returns the corresponding

Algorithm 2: Recursive function to calculate the minimum cost of the system after time t, ending with A nodes

Function `cost`(t, A, L, N_0, Z, m)

Input: Current time interval t, number of nodes A, time series array of predicted load L of length T, machines allocated initially N_0, maximum number of machines available to allocate Z, matrix m of size $T \times Z$ to memoize cost and best series of moves

Output: Minimum cost of the system after time t, ending with A nodes

```
// penalty for constraint violation or insufficient capacity
```

if $t < 0$ *or* $(t = 0$ *and* $A \neq N_0)$ *or* $L[t] >$ `cap`(A) **then return** ∞;

if $m[t,A]$ *exists* **then** `/* check memoized cost */`

  return $m[t,A].cost$;

if $t = 0$ **then** `/* base case */`

  $m[t,A].cost \leftarrow A$;

else `/* recursive step */`

  $B \leftarrow arg\ min_{1 \leq i \leq Z}$(`sub-cost`($t$, i, A, L, N_0, Z, m));

  `// a move must last at least one time interval`

  if $T(B,A) = 0$ **then** $T(B,A) \leftarrow 1$;

  $m[t,A].cost \leftarrow$ `sub-cost`(t, B, A, L, N_0, Z, m);
  $m[t,A].prev\_time \leftarrow t - T(B,A)$;
  $m[t,A].prev\_nodes \leftarrow B$;

return $m[t,A].cost$;

cost. Finally, we come to the recurrence relation. The base case corresponds to $t = 0$, in which we simply return the cost of allocating A machines for one time interval (see Equation (4.1)). The recursive step is as follows: find the cost of the optimal series of moves ending with $B \rightarrow A$, for all B, and choose the minimum. There is one caveat for the case when $B = A$. Since the time and cost of the move are both 0, we need to artificially make the move last for one time step, with a resulting cost of B. This corresponds to the "do nothing" move.

Algorithm 3 finds the cost of the optimal series of moves ending at a given time t with the final move from B to A machines. It first adjusts the time and cost of the move for the case when $B = A$, as described previously for Algorithm 2. Next it checks that the final move from $B \rightarrow A$ would not need to start in the past. Finally, it checks that

Algorithm 3: Recursive function to calculate the minimum cost of the system after time t, with the last move going from B to A nodes

Function sub-cost(t, B, A, L, N_0, Z, m)

Input: Current time interval t, number of machines B before last move, number of machines A after last move, time series array of predicted load L of length T, machines allocated initially N_0, maximum number of machines available to allocate Z, matrix m of size $T \times Z$ to memoize cost and best series of moves

Output: Minimum cost of the system after time t, with the last move going from B to A nodes

```
// a move must last at least one time interval
```
if $T(B,A) = 0$ **then** $T(B,A) \leftarrow 1, C(B,A) \leftarrow B$;

start-move $\leftarrow t - T(B,A)$;

if start-move < 0 **then**

```
    // this reconfiguration would need to start in the past
```
    return ∞;

for $i \leftarrow 1$ **to** $T(B,A)$ **do**

    load $\leftarrow L[$ start-move $+ i]$;

    if load $>$ eff-cap(B, A, $i/T(B,A)$) **then**

```
        // penalty for insufficient capacity during the move
```
        return ∞;

return cost(start-move, B, L, N_0, Z, m) $+ C(B,A)$;

for every time interval during the move from $B \rightarrow A$, the predicted load never exceeds the *effective capacity* of the system, which is the capacity of the system while a reconfiguration is ongoing. We will describe how to compute effective capacity in Section 4.2.4. If all of these checks succeed, it makes a recursive call to Algorithm 2 and returns the cost of the full series of moves.

4.2.4 Characterizing Data Migrations

In order to find an optimal series of moves, the previous algorithms must evaluate individual moves to find the optimal choice at different points in time. This section provides the tools to perform such an evaluation. There are four key questions that must be answered to determine the best choice for a move:

1. How to schedule data transfers in a move?

2. How long does a given reconfiguration take?
3. What is the cost of the system during reconfiguration?
4. What is the capacity of the system to execute transactions during reconfiguration?

The answers to the last three questions correspond to finding expressions for three functions used in Section 4.2.3, respectively: $T(B,A)$, $C(B,A)$, and `eff-cap`. We answer these questions next.

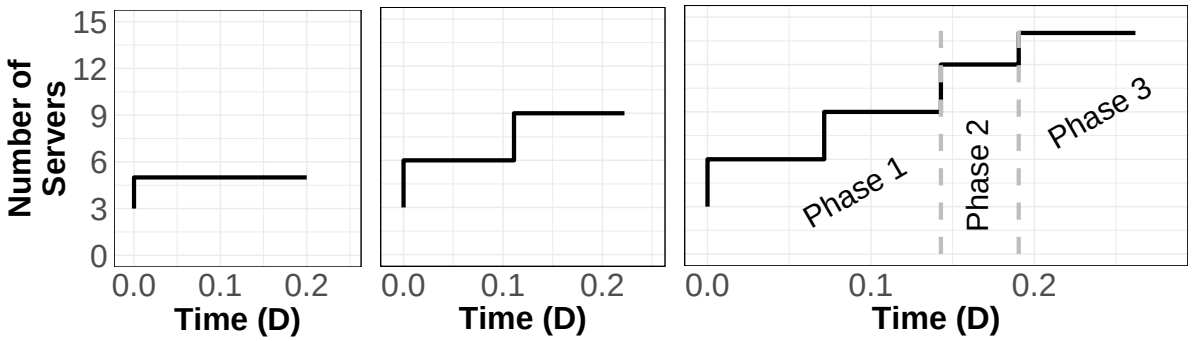
Executing a Move

In order to model moves it is necessary to understand the way they are executed. An important requirement is that at the beginning and end of every move, all servers always have the same amount of data. So initially B machines each have $1/B$ of the data, and at the end A machines each have $1/A$ of the data. Since we consider uniform workloads (see Section 4.2.2), spreading out the data evenly is best for load balancing.

Another important aspect in a move is the degree of parallelism that we can achieve during data migrations. We use a single thread at each partition for transferring data during reconfiguration. This limits the amount of resources used for a move and thus minimizes its performance disruption. If there are fewer partitions sending data than receiving during a reconfiguration, each partition that is sending data during a reconfiguration communicates with exactly one partition receiving data at all times. This minimizes the length of the move by fully utilizing sender partitions, which are fewer than receiver partitions. If there are fewer receiver partitions the opposite holds: each receiver partition communicates with exactly one sender partition at all times. Given these design choices, we can now determine the maximum amount of parallel data transfers that can occur when scaling from B machines before to A machines after, with P partitions per machine as:

$$max_{\|} = \begin{cases} 0 & \text{if } B = A \\ P * min(B, A-B) & \text{if } B < A \\ P * min(A, B-A) & \text{if } B > A \end{cases} \tag{4.2}$$

We are now ready to describe how moves are performed. In the following exposition,



(a) Case 1: $3 \rightarrow 5$ Machines **(b)** Case 2: $3 \rightarrow 9$ Machines **(c)** Case 3: $3 \rightarrow 14$ Machines

Figure 4-4: Servers allocated during parallel migration, scaling out from 3 servers, assuming one partition per server. Time in units of D, the time to migrate all data with a single thread.

Phase 1, Step 1	$1 \rightarrow 4, 2 \rightarrow 5, 3 \rightarrow 6$
	$1 \rightarrow 5, 2 \rightarrow 6, 3 \rightarrow 4$
	$1 \rightarrow 6, 2 \rightarrow 4, 3 \rightarrow 5$
Phase 1, Step 2	$1 \rightarrow 7, 2 \rightarrow 8, 3 \rightarrow 9$
	$1 \rightarrow 8, 2 \rightarrow 9, 3 \rightarrow 7$
	$1 \rightarrow 9, 2 \rightarrow 7, 3 \rightarrow 8$
Phase 2	$1 \rightarrow 10, 2 \rightarrow 11, 3 \rightarrow 12$
	$1 \rightarrow 11, 2 \rightarrow 12, 3 \rightarrow 10$
Phase 3	$1 \rightarrow 12, 2 \rightarrow 13, 3 \rightarrow 14$
	$1 \rightarrow 13, 2 \rightarrow 14, 3 \rightarrow 11$
	$1 \rightarrow 14, 2 \rightarrow 10, 3 \rightarrow 13$

Table 4.1: Schedule of parallel migrations when scaling from 3 machines to 14 machines.

we will consider the specific case of scale out, since the scale in case is symmetrical. For simplicity and without loss of generality, we will assume one partition per server. Moves are scheduled such that the system makes full use of the maximum available parallelism given by Equation (4.2). In addition, moves add new servers as late as possible in order to minimize the cost while the move is ongoing.

When executing moves, there are three possible strategies that P-Store uses to achieve the aforementioned goals, exemplified in Figure 4-4. The first strategy is used when B is greater than or equal to the number of machines that need to be added (see Figure 4-4a). In this case, all new machines are added at once and receive data in parallel, while sender machines rotate to send them data.

In the second case, the number of new machines is a perfect multiple of B, so blocks of

B machines will be allocated at once and simultaneously filled. This allows for maximum parallel movement while also allowing for just-in-time allocation of machines that are not needed right away (see Figure 4-4b).

The third case is the most complex because the move is broken into three phases (see Figure 4-4c). The purpose of the three phases is to keep the sender machines fully utilized throughout the whole reconfiguration, thus minimizing the length of the move. Table 4.1 reports all the sender-receiver pairs in the example of Figure 4-4c.

During the first phase, the system goes through a sequence of steps where new servers are added in blocks of B at a time. In each step, each of the original B servers sends data to every other new server, in a round robin manner.

To see why we need two additional phases, consider again the example of Table 4.1. After the first two steps of phase one, the system has grown to 9 servers. Executing another step of phase one would add another block of 3 servers (since $B = 3$), bringing the system to 12 servers. With only 2 servers left to be added, it would be impossible to make use of all 3 sender servers in the original group, since each receiver partition can communicate only with one sender partition at a time (as described above, this restriction is in place to minimize performance disruption). Accordingly, three rounds of migration would be required in order for all three sender servers to send data to each of the two receivers. To avoid this problem, the migration algorithm activates phase two before the next block of B servers is added. During phase two, the B sender servers send data to B new receiver servers, but they are filled only partly (see Table 4.1). By the end of phase two, each sender server has communicated with only two of the three new receiver servers. Finally, servers 13 and 14 are added during phase three. Because the last batch of B servers were not completely filled in phase two, all the B sender servers can send data in parallel. This enables the full reconfiguration to complete in the 11 rounds shown in Table 4.1, while minimizing overhead on each partition throughout. Without the three distinct phases, the reconfiguration shown would require at least 12 rounds.

Time for a Move

After detailing how a move is scheduled, we are ready to calculate the time $T(B,A)$ that it takes to move from B to A servers. Recall that D is the time it takes to move the entire database using a single thread, as defined in Section 4.2.1. We have discussed previously that moves are scheduled to always make full use of the maximum parallelism given by Equation (4.2). Therefore, the entire database can be moved in time $D/max_{\|}$. If we consider the actual fraction of the database that must be moved to scale from B machines to A machines, we obtain that the reconfiguration time is:

$$T(B,A) = \begin{cases} 0 & \text{if } B = A \\ \frac{D}{max_{\|}} * (1 - \frac{B}{A}) & \text{if } B < A \\ \frac{D}{max_{\|}} * (1 - \frac{A}{B}) & \text{if } B > A \end{cases} \tag{4.3}$$

Cost of a Move

As defined in Equation (4.1), cost depends on the number of machines allocated over time. Therefore, we define the cost of a reconfiguration as follows:

$$C(B,A) = T(B,A) * \texttt{avg-mach-alloc}(B,A) \tag{4.4}$$

where $T(B,A)$ is the time for reconfiguration from Equation (4.3) and $\texttt{avg-mach-alloc}(B,A)$ is the average number of machines allocated during migration, as defined in Algorithm 4.

Algorithm 4 takes into consideration that machine allocation is symmetric for scale-in and scale-out. The important distinction between the starting and ending cluster sizes, therefore, is not before/after but larger/smaller. And the delta between the larger and smaller clusters is equal to the number of machines receiving data from the smaller cluster when scaling out, or the number of machines sending data to the smaller cluster when scaling in. These values are assigned to l, s and Δ in the first few lines of Algorithm 4. The next line assigns to r the remainder of dividing Δ by s, which will be important later in the algorithm.

Algorithm 4: Calculate the average number of machines that must be allocated during the move from B to A machines with parallel migration

Function `avg-mach-alloc(B, A)`

Input: Machines before move B, machines after move A

Output: Average number of machines allocated during the move

```
// Machine allocation symmetric for scale-in and scale-out.
// Important distinction is not before/after but larger/smaller.
```
$l \leftarrow max(B,A)$; // larger cluster
$s \leftarrow min(B,A)$; // smaller cluster
$\Delta \leftarrow l - s$; // delta
$r \leftarrow \Delta \% s$; // remainder

```
// ================================================================
// Case 1: All machines added or removed at once
// ================================================================
```
if $s \geq \Delta$ **then return** l;

```
// ================================================================
// Case 2: Δ is multiple of smaller cluster
// ================================================================
```
if $r = 0$ **then return** $(2s+l)/2$;

```
// ================================================================
// Case 3: Machines added or removed in 3 phases
// ================================================================

// Phase 1: N1 sets of s machines added and filled completely
```
$N_1 \leftarrow \lfloor \Delta / s \rfloor - 1$; // number of steps in $phase_1$
$T_1 \leftarrow s/\Delta$; // time per step in $phase_1$
$M_1 \leftarrow (s+l-r)/2$; // average machines in $phase_1$
$phase_1 \leftarrow N_1 * T_1 * M_1$;

```
// Phase 2: s machines added and filled r/s fraction of the way
```
$T_2 \leftarrow r\ /\Delta$; // time for $phase_2$
$M_2 \leftarrow l - r$; // machines in $phase_2$
$phase_2 \leftarrow T_2 * M_2$;

```
// Phase 3: r machines added and remaining machines filled
// completely
```
$T_3 \leftarrow s/\Delta$; // time for $phase_3$
$M_3 \leftarrow l$; // machines in $phase_3$
$phase_3 \leftarrow T_3 * M_3$;

return $phase_1 + phase_2 + phase_3$

Given these definitions, the algorithm considers the three cases discussed in Section 4.2.4. In the first case, the size of the smaller cluster is greater than or equal to Δ, which means that all new machines must be allocated (or de-allocated) at once in order to allow for maximum parallel movement. In the second case, Δ is a perfect multiple of the smaller cluster, so blocks of s machines will be allocated (or deallocated) at once and simultaneously filled (or emptied). Thus, the average number of machines allocated is $(2s+l)/2$. In the third case we have three phases, and the number of servers added in each phase is shown in Algorithm 4.

Effective Capacity of System During Reconfiguration

Finally we calculate the effective capacity of the system during a reconfiguration. The total capacity of N machines in which data is evenly distributed is defined as follows:

$$\mathsf{cap}(N) = Q * N \tag{4.5}$$

During a reconfiguration, however, data is not evenly distributed. Assume that a node n keeps a fraction f_n of the total database, where $0 \leq f_n \leq 1$. Since we consider uniform workloads, node n receives a fraction f_n of the load, which is $\mathsf{cap}(N) * f_n$ when the system is running at full capacity. The total load on the system cannot be so large that the maximum capacity $\hat{Q}$ of a server is exceeded. Therefore, to account for workload variability, we have that $Q \geq \mathsf{cap}(N) * f_n$ and:

$$\mathsf{cap}(N) \leq Q/f_n \quad \forall n \in \{n_1 \ldots n_N\} \tag{4.6}$$

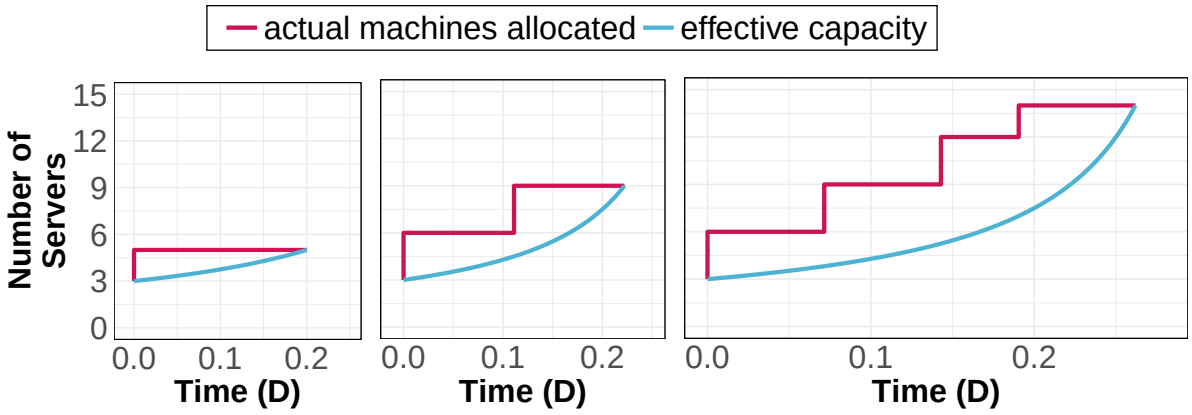
This implies that the maximum capacity of the system is determined by the server having the largest f_n, i.e., the largest fraction of the database. We can thus define the effective capacity of the system after a fraction f of the data is moved during the transition from B machines to A machines as:

$$\texttt{eff-cap}(B,A,f) = \begin{cases} \texttt{cap}(B) & \text{if } B = A \\ \texttt{cap}(1/(\frac{1}{B} - f * (\frac{1}{B} - \frac{1}{A}))) & \text{if } B < A \\ \texttt{cap}(1/(\frac{1}{B} + f * (\frac{1}{A} - \frac{1}{B}))) & \text{if } B > A \end{cases} \tag{4.7}$$

Let us consider each case individually. The first case is self-explanatory; no data is moving. The second case applies to scaling out, where B nodes send data to $(A-B)$ new machines. Throughout reconfiguration, capacity is determined by the original B machines, which each initially have $1/B$ of the data. After reconfiguration, they will have $1/A$ of the data, so as fraction f of the data is moved to the new machines, each of the B machines now has $(1/B - f * (1/B - 1/A))$ of the data. The inverse of this expression corresponds to the number of machines in an evenly-loaded cluster with equivalent capacity to the current system, and Equation (4.5) converts that machine count to capacity. The third case in Equation (4.7) applies to scaling in and follows a similar logic to the second case. Here, $(B-A)$ of the original machines send data to the remaining A machines, each of which start with $1/B$ of the data. After reconfiguration they will have $1/A$ of the data, so as fraction f of the data is moved, each of the A machines now has $(1/B + f * (1/A - 1/B))$ of the data. Passing the inverse of this expression to Equation (4.5) returns the capacity.

This equation assumes that the workload is uniformly distributed across the range of keys. It is incorrect in the case of high skew, because a machine may be overloaded even though it contains less data than other machines. In Section 4.6.1, we show that although the B2W workload is skewed, this skew can be easily masked with a good hash function. Modifying Equation (4.7) for workloads with high skew is left as future work.

To illustrate why it is important to take the effective capacity into account when planning reconfigurations, Figure 4-5 shows the effective capacity at each point during the different migrations presented at the beginning of this section. For a small change such as Figure 4-5a, the effective capacity is close to the actual capacity, and it may not make a difference for planning purposes. But for a large reconfiguration such as Figure 4-5c, the effective capacity is significantly below the actual number of machines allocated. This fact must be taken into account when planning reconfigurations to avoid underprovisioning.



(a) Case 1: 3 → 5 Machines **(b)** Case 2: 3 → 9 Machines **(c)** Case 3: 3 → 14 Machines

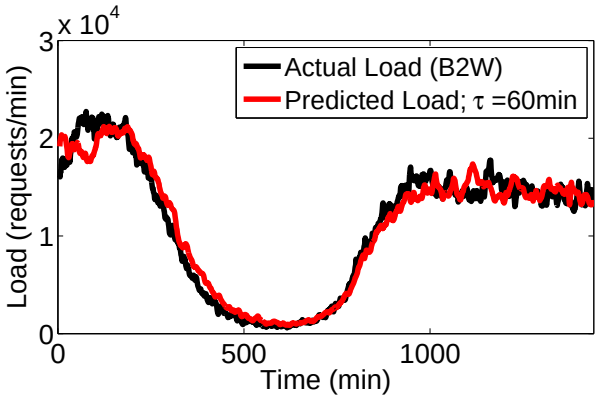
Figure 4-5: Servers allocated and effective capacity during parallel migration, scaling out from 3 servers, assuming one partition per server. Time in units of D, the time to migrate all data with a single thread.

Algorithm 3 performs this function in our Predictive Elasticity Algorithm.

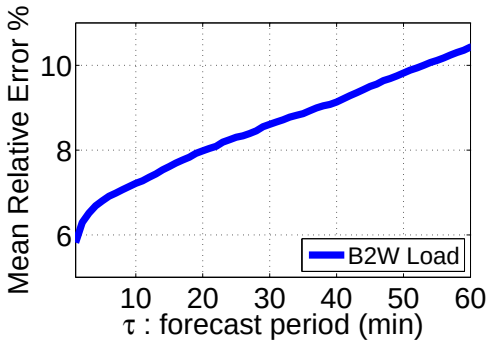
4.3 Load Time-Series Prediction

The decision about how and when to reconfigure requires an accurate prediction of the aggregate workload. In this section, we discuss the time-series techniques used for accurately modeling and predicting the aggregate load on the system. P-Store continuously monitors the load it needs to serve. As shown in Chapter 1, real-world online retail traffic exhibits a strong diurnal pattern (recall the B2W load depicted in Figure 1-1), which is attributed to the customers' daily habits and natural tendencies to shop during specific times of the day. However, we also find that there is variability on a day-to-day basis due to many factors from seasonality of demand to occasional advertising campaigns.

Capturing these short- and long-term patterns when modeling the load is critical for making accurate predictions. And as previously discussed, database reconfiguration usually requires several minutes, therefore prediction of load changes must be done at that scale. To this end, we exploit *auto-regressive* (AR) prediction models which are capable of capturing time-dependent correlations in the data. More precisely, we use *Sparse Periodic Auto-Regression* (SPAR) [17]. Informally, SPAR strives to infer the dependence of the load on long-term periodic patterns as well as short-term transient effects. Therefore, SPAR



(a) 60-minute ahead SPAR predictions during a 24-hour period.



(b) Prediction accuracy vs. forecasting period τ.

Figure 4-6: Evaluation of SPAR's predictions for B2W.

provides a good fit for the database workloads that P-Store is designed to serve, since the AR component captures the observed auto-correlations in the load intensity over time (e.g. due to diurnal trends), and the sparse-periodic component captures longer-term seasonal periods (e.g. due to weekly or monthly trends).

We now discuss fitting SPAR to the load data. We measure the load at time slot t by the number of requests per slot. Here each slot is 1 minute, so $T = 1440$ slots per day. In SPAR we model load at time $t+\tau$ based on the periodic signal at that time of the day and the offset between the load in the recent past and the expected load:

$$y(t+\tau) = \sum_{k=1}^{n} a_k y(t+\tau-kT) + \sum_{j=1}^{m} b_j \Delta y(t-j) \tag{4.8}$$

where n is the number of previous periods to consider, m is the number of recent load measurements to consider, a_k and b_j are parameters of the model, $0 \leq \tau < T$ is a forecasting period (how long in the future we plan to predict) and

$$\Delta y(t-j) = y(t-j) - \frac{1}{n}\sum_{k=1}^{n} y(t-j-kT)$$

measures the offset of the load in the recent past to the expected load at that time of the day. Parameters a_k and b_j are inferred using linear least squares over the training dataset used to fit the model.

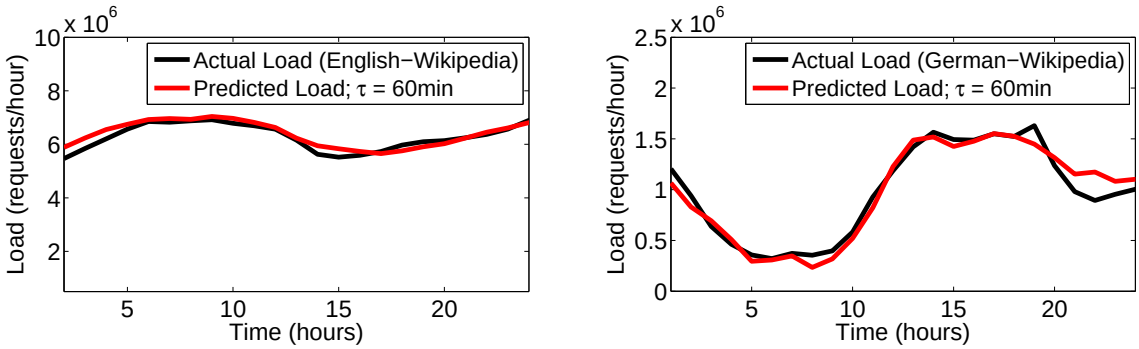
SPAR predictions for B2W: We now analyze the utility of using SPAR to model and predict the aggregate load in B2W. B2W provided us with several months' worth of load traces (Section 4.5 provides more details about the data). We used the first 4 weeks of the data to train the SPAR model. After examining the quality of our predictor under different values for the number of previous periods n and the number of recent load measurements m, we find that setting $n = 7$ and $m = 30$ is a good fit for our dataset. This means we use the previous week for the periodic prediction, and the offset from the previous 30 minutes to indicate of how different current load is from the 'average' load at that time of the day.

To demonstrate the accuracy of our SPAR predictor, in Figure 4-6a we depict the actual B2W load and the SPAR predictions for a 24-hour period (outside of the training set), when using a forecasting window of $\tau = 60$ minutes. We also report the average prediction accuracy as a function of forecasting period τ in Figure 4-6b; the mean relative error (MRE) measures the deviation of the predictions from the actual data. We find that the prediction accuracy decays gracefully with the granularity of the forecasting period τ.

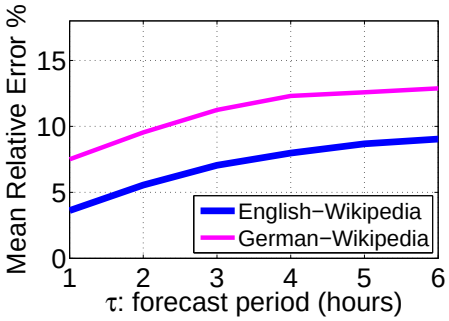
SPAR predictions for less-periodic loads: To better understand if SPAR's high-quality predictions for B2W's periodic load can be obtained for other Internet workloads with less periodic patterns and varying degrees of predictability, we examined traces from *Wikipedia* for their per-hour page view statistics [34]. We focus on the page requests made to the two most popular editions, the English-language and German-language Wikipedias [98].

Similar to our analysis for B2W, we trained SPAR using 4 weeks of Wikipedia traces from July, 2016 (separately for each language trace), then evaluated SPAR's predictions using data from August, 2016. The results in Figures 4-7a and 4-7b show that SPAR is able to accurately model and predict the hourly Wikipedia load for both languages. Even for the less predictable German-language load, the error remains under 10% for predicting up to two hours into the future, and within 13% only for forecast windows as high as 6 hours.

Discussion: What's a good forecast window? Note that the forecast window τ only needs to be large enough so that the first move returned by the dynamic programming algorithm described in Section 4.2 is correct; by the time the first reconfiguration completes, the predictions may have changed, so the dynamic program must be re-run anyway. But in order for the first move to be correct, τ must be at least $2D/P$, the maximum length of



(a) 60-minute ahead SPAR predictions during a 24-hour period.



(b) Prediction accuracy vs. forecasting period τ.

Figure 4-7: Evaluation of SPAR's predictions for another workload with different periodicity and predictability degrees: Wikipedia's per-hour page requests.

time needed for *two* reconfigurations with parallel migration. This allows the algorithm to ensure that the first move will not leave the system in an underprovisioned state for subsequent moves (e.g., if it plans a "scale in" move, it knows there will be time to scale back out in advance of any predicted load spikes). This typically means a value of τ in the range of tens of minutes. Figures 4-6 and 4-7 show that SPAR is sufficiently accurate for such values of τ, with error rates of under 10%.

We have explored other time-series models, such as a simple AR model and an auto-regressive moving-average (ARMA) model. Overall, we find that AR-based models work well, but that SPAR usually produces the most accurate predictions under different workloads (as it captures different trends in the data). Figure 4-8 shows that SPAR's prediction error on the B2W workload is lower than that of both the AR and ARMA models, especially as the forecast window increases. Based on these results, the rest of this chapter focuses on the rich dataset from B2W as a large-scale, real-world representative workload for online OLTP applications, and on SPAR as a load prediction model.

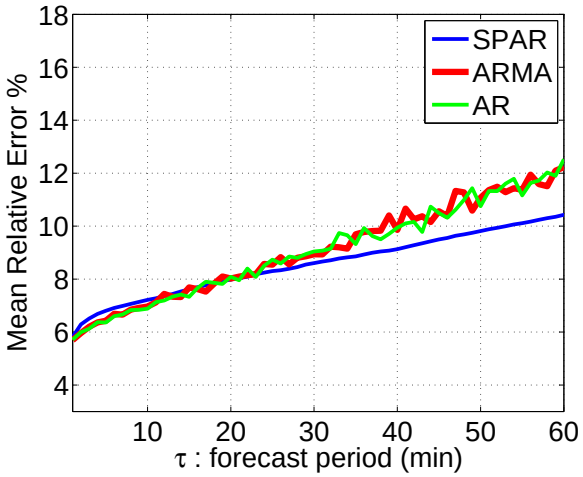


Figure 4-8: Comparison of SPAR prediction accuracy with other auto-regressive models on the B2W workload

4.4 Putting It All Together

As described in Chapter 2, P-Store's techniques are general and can be applied to any partitioned DBMS, but our first P-Store implementation is based on H-Store and its live migration system, Squall.

The P-Store system combines all the previous techniques for time series prediction, determination of when to scale in or out, and how to schedule the migrations. We have created a "Predictive Controller" which handles online monitoring of the system and calls to the components that implement each of these techniques. For convenience, we call these components the Predictor, the Planner and the Scheduler, respectively. To obtain measurements of the aggregate load in H-Store, the Predictive Controller calls H-Store's system stored procedures.

P-Store has an active learning system. If training data exists, parameters a_k and b_j in Equation (4.8) can be learned offline. Otherwise, P-Store constantly monitors the system over time and can actively learn the parameter values. The Predictor component uses Equation (4.8) and the fitted parameter values to make online predictions based on current measurements of H-Store's aggregate load.

As soon as P-Store starts running, the Predictive Controller begins monitoring the system and measuring the load. When enough measurements are available, it makes a call to

the Predictor, which returns a time series of future load predictions. The Controller then passes this data to the Planner, which calculates the best series of moves.

Given the best series of moves from the Planner, the Controller throws away all but the first (similar to the idea of receding horizon control [65]). It passes this first move along with the current partition plan to the Scheduler, which generates a new partition plan in which all source machines send an equal amount of data to all destination machines, as described in Section 4.2.4. This partition plan is then passed to the external Squall system to perform the migration.

If the Planner calls for a scale-in move, the Controller waits for three cycles of predictions from the Predictor to confirm the scale-in. If after three cycles the Planner still calls for a scale-in, then the move is executed. This heuristic prevents unnecessary reconfigurations that could cause latency spikes.

After a move is complete, the Controller repeats the cycle of prediction, planning, and migration. If at any time the Planner finds that there is no feasible solution to manage the load without disruption, the Scheduler is called to create a partition plan to scale out to the number of machines needed to handle the predicted spike, and Squall is called in one of two ways as described at the end of Section 4.2.3: move data faster and suffer some latency during migration, or move at the regular rate and wait longer to reach the desired capacity.

4.5 B2W Digital Workload

B2W has provided a large, rich dataset that includes logs of every transaction on their shopping cart, checkout and stock inventory databases over a period of several months, as well as the historical CPU utilization and I/O operations on the database servers. The transaction logs include the timestamp and the type of each transaction (e.g., GET, PUT, DELETE), as well as unique identifiers for the shopping carts, checkouts and stock items that were accessed or modified. Since there is some important information not available in the logs (e.g., the contents of each shopping cart), B2W has also provided a dump of all of the shopping carts, checkouts, and stock data from the last year. All data for this project has been anonymized by B2W to eliminate any sensitive customer information,

Stock Inventory

sku	description	available	reserved	purchased
123456	Harry Potter and the…	97	2	53
111111	Maytag front loadin…	43	0	13
…	…	…	…	…

Shopping Cart

cart_id	cust_id	timestamp
abcdef	000001	Aug 2, 2016 10:05:34
ababab	000002	Aug 5, 2016 11:12:13
…	…	…

Cart Lines

cart_id	sku	price
abcdef	123456	$10.99
abcdef	111111	$599.99
…	…	…

Checkout

cart_id	checkout_id	credit_card_no	expiration
abcdef	abcdefghi	1111111111111111	11/18
bcbcbc	bcbcbcdcdc	2222222222222222	07/17
…	…	…	…

Figure 4-9: Simplified database for the B2W H-Store benchmark

but otherwise it is identical to the data in production. Joining the unique identifiers from the log data with the keys in the database dump thus allows us to infer almost everything about each transaction, meaning we can effectively replay the transactions starting from any point in the logs. This allows us to run H-Store with the same workload running in B2W's production shopping cart, checkout and stock databases.

To model B2W's workload in H-Store, we have implemented a benchmark driven by their traces. This benchmark includes nearly all the database operations required to run an online retail store, from adding and removing items in customers' shopping carts, to collecting payment data for checkout. A simplified database is shown in Figure 4-9, and a list of the transactions is shown in Table 4.2. When a customer tries to add an item to their cart through the website, GetStockQuantity is called to see if the item is available, and if so, AddLineToCart is called to update the shopping cart. At checkout time, the system attempts to reserve each item in the cart, calling ReserveStock on each item. If a given item is no longer available, it is removed from the shopping cart and the customer is notified. The customer has a chance to review the final shopping cart before they agree to the purchase.

Transaction	Description
AddLineToCart	Add a new item to the shopping cart, create the cart if it doesn't exist yet
DeleteLineFromCart	Remove an item from the cart
GetCart	Retrieve items currently in the cart
DeleteCart	Delete the shopping cart
GetStock	Retrieve the stock inventory information
GetStockQuantity	Determine availability of an item
ReserveStock	Update the stock inventory to mark an item as reserved
PurchaseStock	Update the stock inventory to mark an item as purchased
CancelStockReservation	Cancel the stock reservation to make an item available again
CreateStockTransaction	Create a stock transaction indicating that an item in the cart has been reserved
ReserveCart	Mark the items in the shopping cart as reserved
GetStockTransaction	Retrieve the stock transaction
UpdateStockTransaction	Change the status of a stock transaction to mark it as purchased or cancelled
CreateCheckout	Start the checkout process
CreateCheckoutPayment	Add payment information to the checkout
AddLineToCheckout	Add a new item to the checkout object
DeleteLineFromCheckout	Remove an item from the checkout object
GetCheckout	Retrieve the checkout object
DeleteCheckout	Delete the checkout object

Table 4.2: Operations from the B2W H-Store benchmark

Although the data provided by B2W is proprietary, the H-Store benchmark containing the full database schema and transaction logic is not. The benchmark is open-source and available on GitHub for the community to use [91].

Using B2W's Workload to Evaluate P-Store: To evaluate P-Store, we ran the B2W benchmark described above. The cart and checkout databases have a predictable access pattern (recall Figure 1-1) due to the daily habits of B2W's customers. The stock database is more likely to be accessed by internal B2W processes, however, causing a spiky, unpredictable access pattern, which is possibly related to the deliveries that replenish the stock. There may be predictive models that would be appropriate for this spiky workload, but

we leave such a study for future work. Therefore, our evaluation considers only data and transactions from the cart and checkout databases. This is consistent with the deployment used in production at B2W: the stock data is stored in a different database, on a different cluster of machines from the cart and checkout data. The business logic involving multiple data sources happens at the application layer, not at the database layer.

When replaying the original cart and checkout transactions, we make a couple of modifications to enable us to experimentally demonstrate our proactive elasticity algorithms with H-Store and Squall. First, we increase the transaction rate by $10\times$ so that we can experience the workload variability of a full day in just a few hours. This allows us to demonstrate the performance of P-Store over several days within a reasonable experimental timeframe. Second, we add a small delay in each transaction to artificially slow down execution. We do this because H-Store is much faster than the DBMS used by B2W and can easily handle even the accelerated workload with a single server. Slowing down execution allows us to demonstrate the effectiveness of P-Store by requiring multiple servers.

We train our prediction model using 4-weeks' worth of historical B2W data, stored in an analytic database system. The SPAR parameters a_k and b_j from Equation (4.8) are calculated offline using the training data and can be easily updated online periodically, even though we did not implement these periodic updates. The remaining parameters in SPAR are updated online based on current load information extracted from H-Store.

4.6 Evaluation

To evaluate P-Store we run the B2W workload described in the previous section. All of our experiments are conducted on an H-Store database with 6 partitions per node deployed on a 10-node cluster running Ubuntu 12.04 (64-bit Linux 3.2.0), connected by a 10 Gb switch. Each machine has four 8-core Intel Xeon E7-4830 processors running at 2.13 GHz with 256 GB of DRAM.

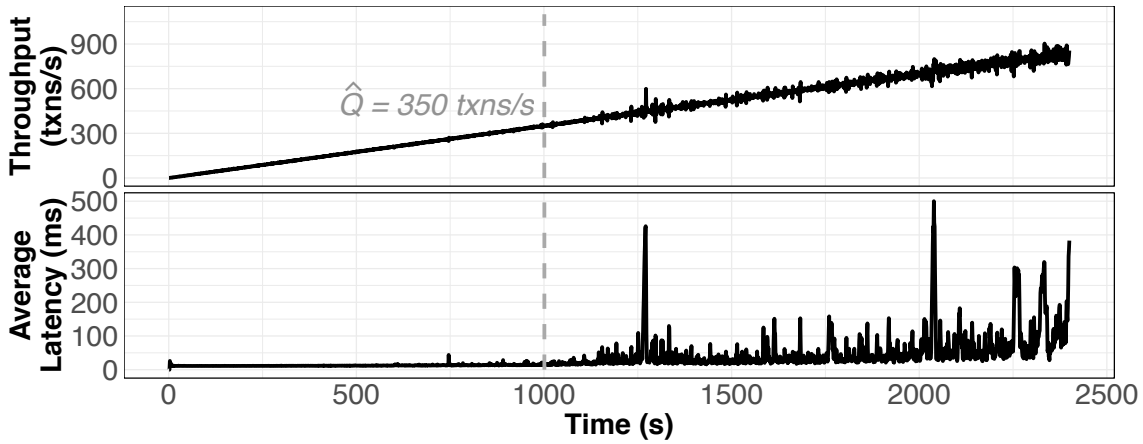


Figure 4-10: Increasing throughput on a single machine. Gray line indicates maximum throughput $\hat{Q}$

4.6.1 Parameter Discovery

Before running P-Store, we need to understand certain properties of the B2W workload and its performance on our system. In particular, we must confirm that the workload is close to uniform, and we must determine the target and maximum throughput per machine Q and $\hat{Q}$ and the time to migrate the whole database D as described in Section 4.2.1.

In the B2W workload, each shopping cart and checkout key is randomly generated, so there is minimal skew in transactions accessing the cart and checkout databases. Furthermore, after hashing the keys to partitions with MurmurHash 2.0 [45], we found that the access pattern and data distribution are both relatively uniform. In particular, with 30 partitions over a 24-hour period, the most-accessed partition receives only 10.15% more accesses than average, and the standard deviation of accesses across all partitions is 2.62% of the average. The partition with the most data has only 0.185% more data than average, and the standard deviation is 0.099% of the average. This level of skew is not even close to the skew described in [92, 83], in which 40% or more of the transactions could be routed to a single partition. Therefore, the assumption that we have a uniform database workload is reasonable.

To discover the values for Q and $\hat{Q}$, we run a rate-limited version of the workload with a single server and identify the transaction rate at which the single server can no longer keep up. As shown in Figure 4-10, for the B2W workload running on an H-Store cluster

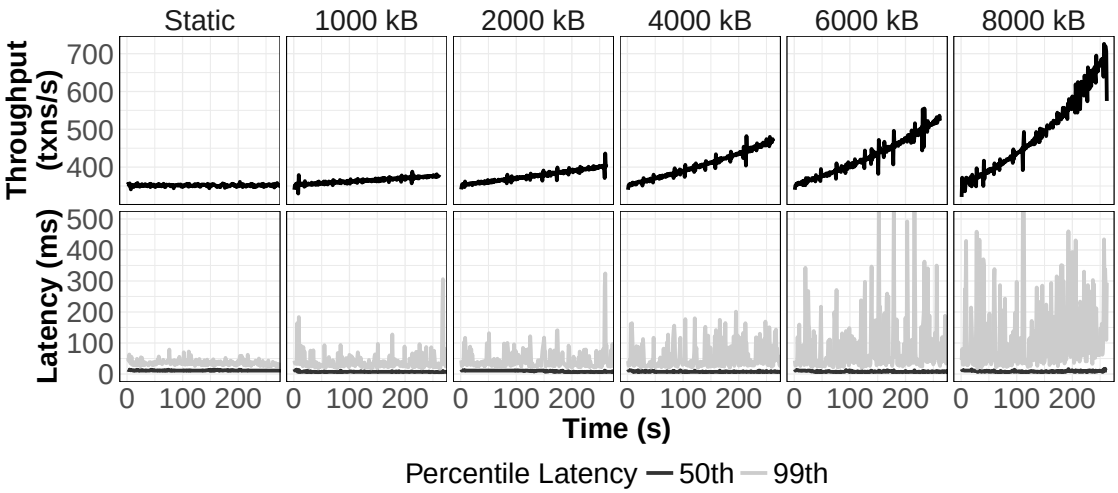


Figure 4-11: 50th and 99th percentile latencies when reconfiguring with different chunk sizes compared to a static system. Total throughput varies so per-machine throughput is fixed at $\hat{Q}$.

with 6 partitions per server, this happens at 438 transactions per second. As described in Section 4.2.1, we set $\hat{Q}$ to 80% of the empirical maximum, or 350 transactions per second. Q is set to 65% of the maximum, or 285 transactions per second.

To discover D, we run the following set of experiments: With an underlying workload of $\hat{Q}$ transactions per second, we start with the data on a single machine and move half of the data to a second machine, tracking the latency throughout migration. We perform this same experiment several times, varying the migration chunk size each time. We also vary the overall transaction rate to ensure that the rate on the source machine stays fixed at $\hat{Q}$, even as data is moved. As shown in Figure 4-11, the 99th percentile latency when moving 1000 kB chunks is slightly larger than that of a static system with no reconfiguration, but still within the bounds of most acceptable latency thresholds. Moving larger chunks causes the reconfiguration to finish faster, but creates a higher risk for latency spikes. In the 1000 kB experiment we moved one half of the entire 1106 MB database of active shopping carts and checkouts in 2112 seconds. Therefore, we set D to 4646 seconds (including the 10% buffer), or 77 minutes. We define the *migration rate* R as the rate at which data is migrated in this setting, which is 244 kB per second[1]. Since P-Store actually performs parallel mi-

[1]A data movement rate of 244 kB per second may seem low given a setting of 1000 kB chunks, but the reported chunk size is actually an upper bound; the actual size of most chunks is much smaller. Squall also spaces the chunks apart by at least 100 ms on average.

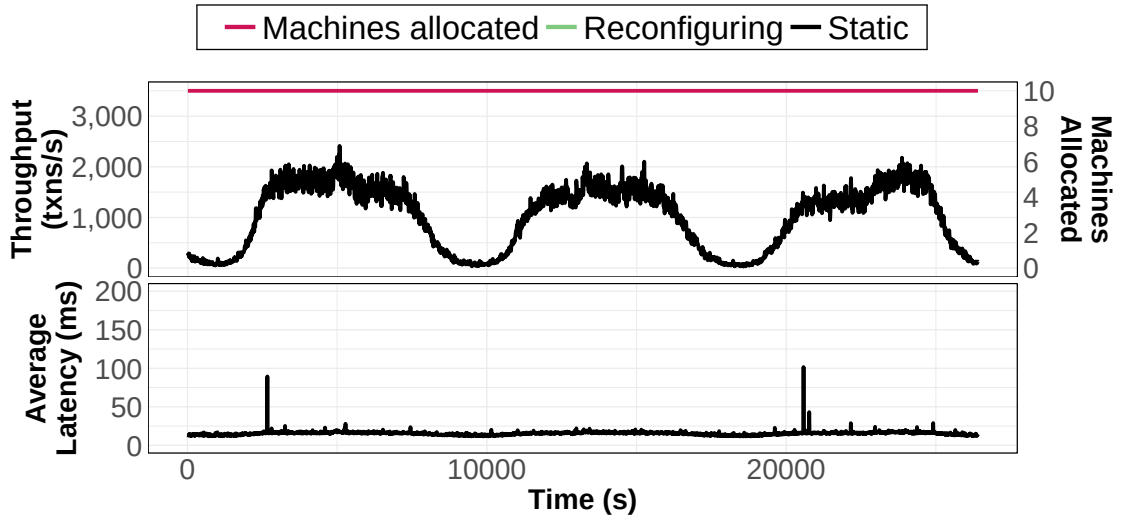
gration and a single migration never moves the entire database, most reconfigurations last between 2 and 7 minutes. The reason Squall takes so long to reconfigure the database is that in order to conform to H-Store's single-threaded execution model and ensure consistency, it must lock both the source and destination partitions and effectively perform a distributed transaction for every data movement (see Section 2.3). Distributed transactions are expensive in H-Store, so Squall spaces them apart in order to minimize impact on transaction latency.[2]

4.6.2 Comparison of Elasticity Approaches

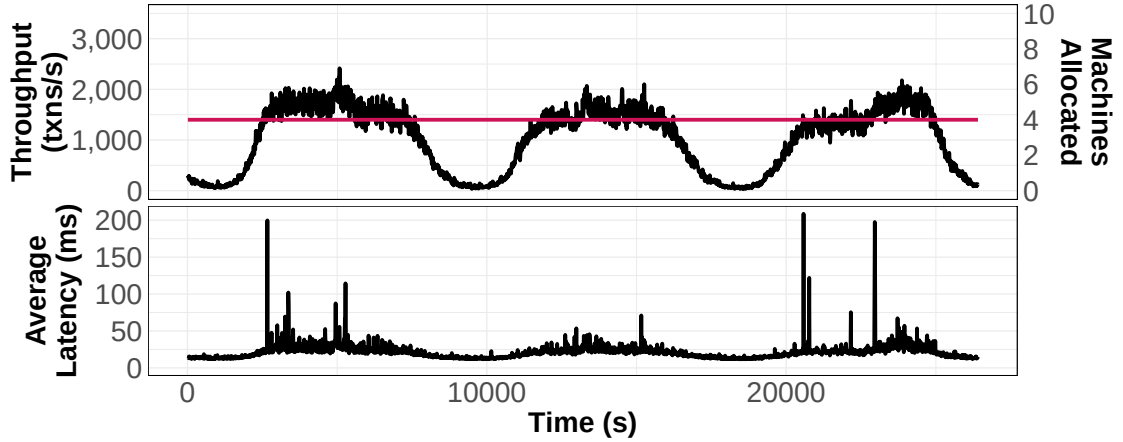
In this section, we compare the performance and resource utilization of several different elasticity approaches. Unless otherwise noted, all of the experiments are run with the B2W benchmark replaying transactions from a randomly chosen 3-day period, which happened to fall in July 2016. With a 10× speedup, this corresponds to 7.2 hours of benchmark time per experiment. For visual clarity, all of the charts show throughput and latency averaged over a 10 second window. To account for load prediction error, we inflate all predictions by 15%.

As a baseline for comparison, we run the benchmark on H-Store with no elasticity. If the number of servers used is sufficient to manage the peak load comfortably, we would expect few high latency transactions but many idle servers during periods of low activity. Figure 4-12a shows this scenario when running the B2W benchmark on a 10-node cluster. Throughput follows the familiar sinusoidal pattern, and average latency remains low, with only two small spikes during the first and third days. Presumably these spikes are caused by transient workload skew (e.g., one partition receives a large percentage of the requests over a short period of time). The red line at the top of the chart shows that with 10 machines allocated and a capacity per machine of $\hat{Q} = 350$ transactions per second, there is plenty of capacity for the offered load. If we reduce the number of servers to 4, the number of idle machines drops but the number of high latency transactions increases (Figure 4-12b).

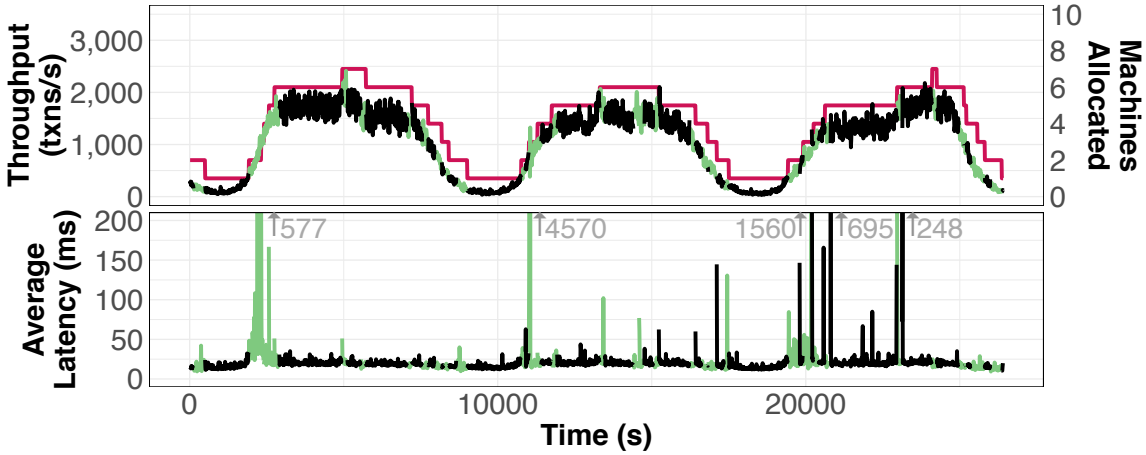
[2]This rate of migration is acceptable for the B2W workload since the database is relatively small. It may not be acceptable for a much larger database, however. Squall's performance is commensurate with its status as an academic prototype; finely tuned commercial systems such as VoltDB [101] are much faster, and would likely decrease P-Store's total migration time.



(a) Throughput and latency over three days with a statically provisioned cluster with 10 machines

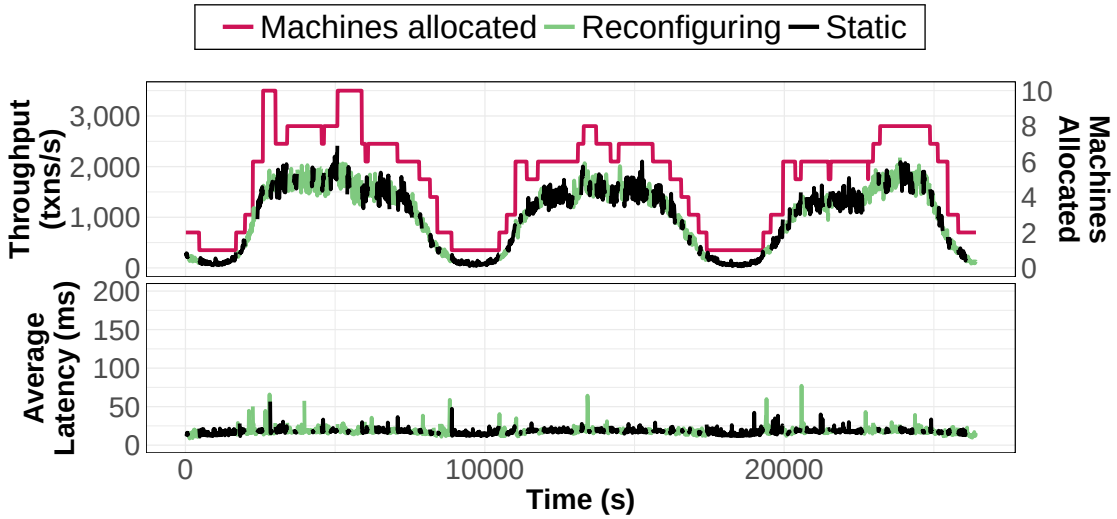


(b) Throughput and latency over three days with a statically provisioned cluster with 4 machines

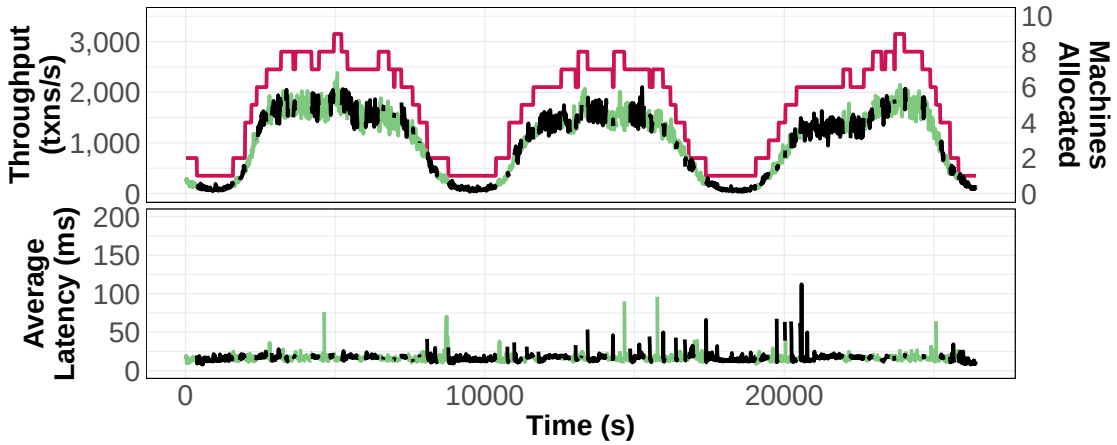


(c) Performance of a reactive system with the B2W workload

Figure 4-12: Comparison of elasticity approaches – static and reactive provisioning



(a) Performance of P-Store with the SPAR predictive model



(b) Performance of P-Store with an oracle predictor

Figure 4-13: Comparison of elasticity approaches – predictive provisioning

These latency spikes are unacceptable for companies that require fast response times, so the static approach with provisioning for peak load is the one used by most OLTP databases, including B2W's current production system. Although B2W uses fewer than 10 servers for their peak workload today, they anticipate needing more as their business grows, making peak provisioning more and more expensive.

We also run the benchmark with the reactive elasticity technique used by E-Store [92]. We choose E-Store over Clay [83] because in the B2W benchmark each transaction accesses only one partitioning key. Figure 4-12c shows the performance of this technique on the B2W workload. Light green sections of the throughput and latency curves indicate that a reconfiguration is in progress, while black sections indicate a period of no data movement. The red line shows the number of machines allocated at each point in time and the corresponding machine capacity (effective capacity is not shown, but it is close to the full machine capacity). Clearly, this technique reacts to the daily variations in load and correctly reconfigures the system as needed to meet demand. However, the system has high latency at the start of each load increase due to the overhead of reconfiguration in the presence of increasing load.

Finally, we show that P-Store comes closest to solving the problem outlined in Section 4.1. Figure 4-13a shows P-Store with the SPAR predictor ("P-Store SPAR") running on the B2W benchmark. We see many fewer latency spikes than the reactive experiment because P-Store reconfigures the system in advance of load increases and provides more headroom for transient load variations and skew (notice that the red line indicating machine capacity is always above the throughput curve). For comparison, Figure 4-13b shows the performance of the P-Store system when using an oracle predictor that has perfect knowledge of the future. P-Store SPAR actually causes fewer latency spikes than P-Store with the oracle predictor because P-Store SPAR is more conservative and less likely to scale in when there is a slight dip in throughput. As a result, P-Store SPAR also uses slightly *more* machines on average.

Figure 4-14 compares the five different elasticity approaches studied in terms of CDFs of the top 1% of 50th, 95th and 99th percentile latencies measured each second during the experiments shown in Figures 4-12 and 4-13. Curves that are higher and far to the

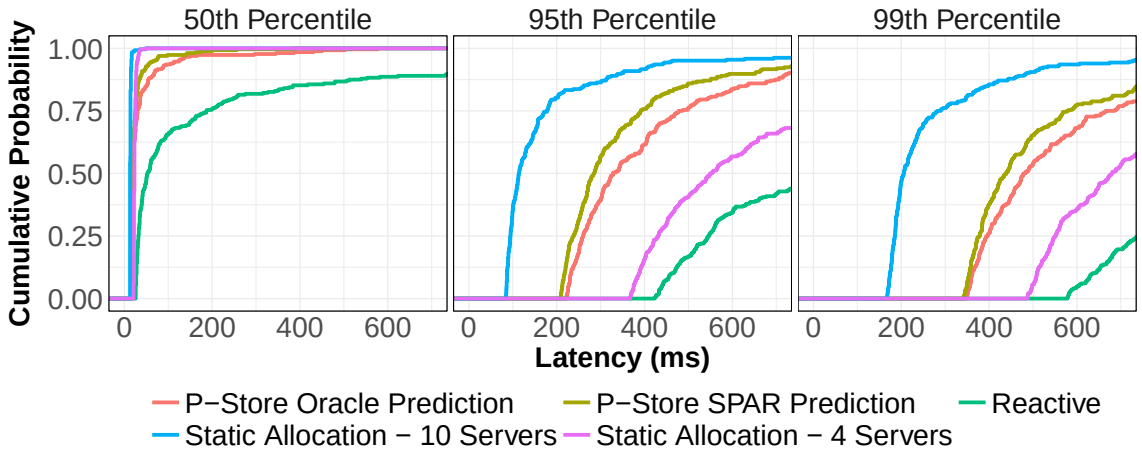


Figure 4-14: Comparison of elasticity approaches in terms of the top 1% of 50th, 95th and 99th percentile latencies.

Elasticity Approach	# Latency Violations 50th %ile	95th %ile	99th %ile	Average Machines Allocated
Static allocation with 10 servers	0	13	25	10
Static allocation with 4 servers	0	157	249	4
Reactive provisioning	35	220	327	4.02
P-Store with SPAR predictor	0	37	92	5.05
P-Store with oracle predictor	2	63	121	4.89

Table 4.3: Comparison of elasticity approaches in terms of number of SLA violations for 50th, 95th and 99th percentile latency, and average machines allocated. SLA violations are counted as total number of seconds with latency above 500 ms.

left are better, because that indicates that latency is generally low. The reactive approach clearly performs the worst in all three plots. Although static allocation with four servers outperforms both P-Store approaches for 50th percentile latency, it is much worse for 95th and 99th percentile latencies. Static allocation with 10 servers performs best in all three plots, but the P-Store approaches are not far behind.

Table 4.3 reports the number of SLA violations as well as the average number of machines allocated during the experiments shown in Figures 4-12 and 4-13. We define SLA violations as the total number of seconds during the experiment in which the 50th, 95th, or 99th percentile latency exceeds 500 ms, since that is the maximum delay that is unnoticeable by users [6]. Static allocation with 10 machines unsurprisingly has the fewest latency violations, but it also has at least 2× more machines allocated than all the other approaches.

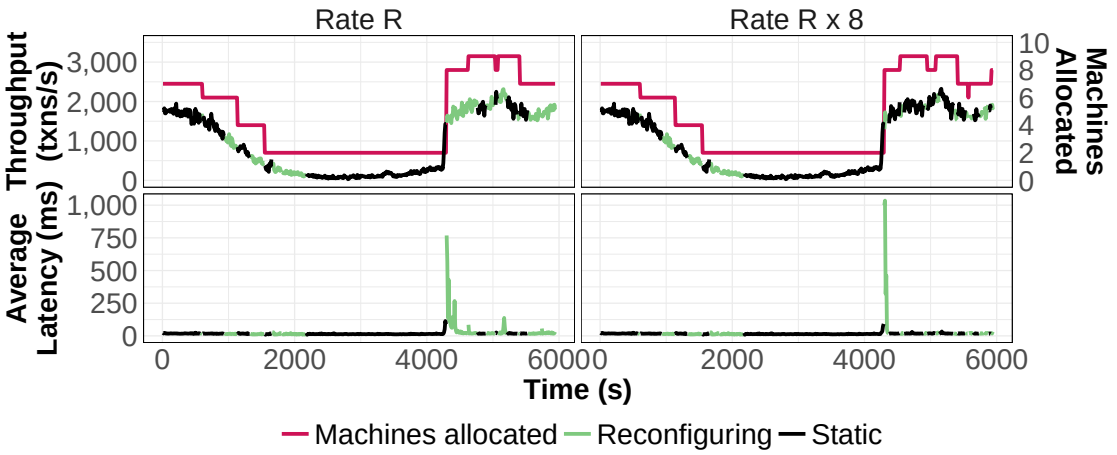


Figure 4-15: Comparison of two different rates of data movement when reacting to an unexpected load spike, under P-Store SPAR.

Static allocation with 4 machines has the fewest number of machines allocated, but it has many SLA violations for the tail latencies. Reactive provisioning performs even worse, with 13× more 99th percentile latency violations than static allocation for peak load. Both versions of P-Store perform well, using about 50% of the resources of peak provisioning, while causing about one third of the latency violations of reactive provisioning. As discussed previously, the SPAR variant of P-Store is more conservative than the one using the oracle predictor and thus leads to fewer latency violations. P-Store has more latency violations than the peak-provisioned system because there is less capacity to handle transient workload skew, particularly when it coincides with data movement. This will be less of a problem when running at normal speed (as opposed to 10× speed), because the system will need to reconfigure less frequently. Users can also configure P-Store to be more conservative in terms of the target throughput per server Q.

The predictive algorithms alone are sufficient and the performance of P-Store matches the version with the oracle predictor as long as there are no unexpected load spikes. When the predictions are incorrect, however, P-Store must do one of the two options described in Section 4.2: continue scaling out at rate R, or increase the rate of migration to scale out as fast as possible. Figure 4-15 compares these two different approaches in the presence of a large unexpected spike during a day in September 2016. When scaling at rate R, the numbers of latency violations for the 50th, 95th, and 99th percentile are 16, 101, and

143, respectively. When scaling at rate $R \times 8$, however, the numbers are 22, 44, and 51. Although the average latency at the start of the load spike is higher when scaling at rate $R \times 8$, the total number of seconds with latency violations is lower.

4.7 Conclusion

This chapter has presented P-Store, an elastic database system that uses predictive modeling for proactive reconfiguration. It forecasts future load on the database using Sparse Periodic Auto-Regression (SPAR), and determines the minimum number of servers needed at each point in the future in order to serve the predicted load. Based on this prediction, P-Store uses a dynamic programming algorithm to plan a series of reconfigurations so that the average number of servers is minimized and capacity always exceeds the predicted load. A live migration system such as Squall then executes each reconfiguration using P-Store's data migration scheduling algorithm, which maximizes efficiency while ensuring that no servers are overloaded. The evaluation shows that on a real online retail workload, P-Store uses 50% fewer servers than static provisioning for peak demand. Although P-Store causes more latency violations than the peak provisioned system, it is a significant improvement over prior elastic approaches, causing 72% fewer latency violations than a reactive elasticity system.

Chapter 5

Related Work

There are three major bodies of work that are relevant to the two elasticity systems presented here. First of all, there are many other DBMSs that support elastic scaling, however none successfully use the two-tiered approach for managing skew proposed by E-Store, and none take advantage of predictive modeling as P-Store does. Second, there is some work on predictive modeling for scalable systems in general, but not specifically applied to an OLTP DBMS. Finally, there is a large body of work on live migration of DBMSs. This thesis uses that work and adds to it by proposing a model to characterize the elapsed time and cost of a reconfiguration, as well as the capacity of the DBMS during reconfiguration.

5.1 Elasticity Techniques

This thesis follows on several previous papers on database elasticity. The E-Store system described here was initially published as a conference paper [92]. As described in Chapter 2, E-Store does not consider distributed transactions, so it is designed for workloads in which most transactions access a single partitioning key (with a "tree schema"). Clay [83] generalizes the E-Store approach to multi-key transactions (non-tree schemas). Rather than moving individual hot tuples, Clay moves "clumps" of hot and cold tuples that are frequently accessed together in order to balance the workload without creating new distributed transactions. Cumulus [31] is another project that, similar to Clay, attempts to minimize distributed transactions through adaptive repartitioning. It currently does not

support elasticity.

Recent work has explored the problem of supporting multi-tenant deployments in cloud-oriented DBMSs. This is exemplified by Salesforce.com's infrastructure that groups applications onto single-node Oracle DBMS installations [104]. Similar work in the Kairos [21] and Zephyr [30] projects pack single-node DBMSs together on multi-tenant nodes. In contrast, this work focuses on elastically provisioning single applications onto multiple nodes in a distributed DBMS.

Essentially all data warehouse DBMSs use hash or range partitioning, and provide some level of on-line reprovisioning. Early work on load balancing by repartitioning for Aster Data could reorganize a range-partitioned database [35]. Later in the 2000s, several NoSQL DBMSs were released that use consistent hashing, popularized in Chord [86], to assign tuples to shared-nothing nodes.

NuoDB [74] and VoltDB [101] are NewSQL DBMSs [9] that partition data across multiple nodes in a computing cluster and support on-line reprovisioning. NuoDB uses physical "atoms" (think disk pages) as their unit of partitioning, while VoltDB uses hash partitioning. A key difference between E-Store and all of these products is that E-Store's two-tier partitioning scheme supports both fine- and coarse-grained tuple assignment, and its tightly-coupled approach balances the overhead of the migration of data and the expected performance improvement after the migration. Although P-Store uses a one-tiered approach, it is differentiated due to the use of predictive modeling.

Since the original E-Store paper was published, Google published a paper indicating that their Spanner database has the capability for automatic load balancing by repartitioning at the level of fine-grained key ranges [12]. This design enables a two-tiered approach such as E-Store's, but the authors do not provide details about their load balancing algorithm or monitoring infrastructure, so it is not clear if they take advantage of this capability.

Hong et al. proposed a method, called *SPORE*, for self-adapting, popularity-based replication of hot tuples [47]. This method mitigates the effect of load imbalance in key-value DBMSs, such as memcached [73]. SPORE does not support ACID semantics nor scaling in/out the number of nodes. It replicates hot keys by renaming them and then this replication is performed randomly without considering underloaded nodes.

Accordion is a one-tiered elasticity controller that explicitly models the effect of distributed transactions on server capacity [82]. As with other one-tier approaches, Accordion relies on a pre-defined set of blocks that can be migrated but are never modified. Accordion is not able to handle situations where hotspots concentrate on a particular block and make it impossible for any server to process the load of that block. By contrast, E-Store's two-tiered approach is able to detect heavily accessed hot tuples within a block, isolate them, and redistribute them to underloaded nodes.

ElasTraS is an elastic and scalable transactional database [23]. ElasTraS utilizes a decoupled storage architecture that separates storage nodes from transaction manager nodes, each of which is exclusively responsible for a data partition. The focus is on fault tolerance, novel system architecture, and providing primitives for elasticity, such as the ability to add and move partitions [24]. However, ElasTraS emphasizes support for multi-tenant databases and transaction execution is limited to a single partition, or transaction manager. Therefore, ElasTraS cannot support databases that must be partitioned across several nodes. Conversely, load-balancing is accomplished by a greedy heuristic that migrates tenants from over-loaded nodes to the least-utilized nodes. Details for loadbalancing and partition splitting are not presented by the authors.

PLP is a partitioning technique that alleviates locking and logging bottlenecks in a shared-memory DBMS [95]. It recursively splits hot data ranges into fixed-size sub-ranges that are distributed among the partitions. This approach works well with hot ranges that are large, but requires many sub-range splits before it is able to isolate single hot tuples. As the number of ranges grows, monitoring costs grow too. PLP continuously monitors the load on each of the newly created sub-ranges, which has a non-negligible performance impact during regular execution. E-Store is focused on repartitioning for distributed DBMSs, and supports scaling in/out as well as load balancing across multiple servers. E-Store normally uses a lightweight monitoring protocol, and turns on more detailed monitoring only when needed and for a short period of time. This makes it possible to immediately isolate hot spots without having to go through multiple repartitioning cycles.

There has been some work on "vertical" (as opposed to "horizontal") scaling, in which the amount of CPU and RAM available to tenants in a single VM can vary over time [94,

103]. This work is complementary to the research presented here, but is not the focus of this thesis.

5.2 Predictive Modeling for Scalable Systems

Many recent papers have modeled cyclic workloads and load spikes for management of data centers, Infrastructure-as-a-Service cloud systems, and web applications [37, 61, 84, 100, 85, 39]. Many of the systems described are elastic and include a control-loop for proactively provisioning resources in advance of load increases. The model for most of these systems is that servers and other cloud resources have some amount of fixed initialization cost, but once initialized they are available to serve requests at full capacity. We study a more complex model of proactive provisioning specific to shared nothing databases, in which the effective capacity of newly allocated servers is limited by the speed of data re-distribution.

There has been some recent work on modeling workloads for elastically scaling databases [27], but it has focused on long-term growth for scientific databases rather than cyclic OLTP workloads. Holze et al. model cyclic database workloads and predict workload changes [46], but they do not use these models to proactively reconfigure the database. PerfEnforce [75] predicts the amount of computing resources needed to meet SLAs for a particular OLAP query workload. It does not take into account the time to scale out to the new configuration, nor the impact on query performance during scaling. ShuttleDB implements predictive elasticity for a multi-tenant Database-as-a-Service system, but unlike our system it only moves entire databases or VMs [13]. It appears that there is no other system that solves the problem that P-Store addresses: proactive scaling for a distributed, highly-available OLTP database.

5.3 Live Migration Techniques

Several live migration techniques have been proposed to move entire databases from one node to another with minimized interruption of service and downtime. Designed for sys-

tems with shared storage, Albatross [24] copies a snapshot of transaction state asynchronously to a destination server. In addition, Slacker [14] is another approach that is optimized for minimizing the impact of migration in multi-tenant DBMSs by throttling the rate that pages are migrated from the source to destination. Zephyr [30] allows concurrent execution at the source and destination during migration, without the use of distributed transactions. Although Zephyr does not require the nodes to be taken off-line at any point, it does require that indexes are frozen during migration. ProRea [81] extends Zephyr's approach, but it instead proactively migrates hot tuples to the destination at the start of the migration.

Previous work has also explored live reconfiguration techniques for partitioned, distributed DBMSs. Wildebeest employed both reactive and asynchronous data migration techniques for a distributed MySQL cluster [51]. In [68] a method is proposed for VoltDB that uses statically defined virtual partitions as the granule of migration. Lastly, as described above, Squall [28] allows fine-grained on-line reconfiguration of partitioned databases. In theory, E-Store can use any of these transport mechanisms; the prototype presented here uses a modified version of Squall since it already supports fine-grained partitioning with H-Store.

There has been a great deal of work on characterizing the cost and time of virtual machine live migration [1, 60, 102]. For live migration of databases, however, cost metrics previously studied include service unavailability, number of failed requests, impact on response time, and data transfer overhead [23]. Previous work on live migration of databases has focused on reducing these sources of overhead while maintaining ACID guarantees [28, 23, 30, 24, 14]. This thesis shows that with careful performance tuning, it is possible to virtually eliminate these sources of overhead for certain workloads.

Chapter 6

Future Work

There are several directions for future research into elastic database systems. Low hanging fruit includes extensions to the existing E-Store and P-Store systems and possible unification of the two systems into one ideal elastic system. Other interesting research directions include comparing the benefits of replication versus partitioning, as well as comparing scale-up approaches to elasticity with scale-out approaches.

6.1 Extensions to E-Store and P-Store

There are several directions for future research on E-Store and P-Store. An obvious path is to unify these two systems into a single system which uses predictive modeling for proactive reconfiguration, but also manages skew with E-Store's two-tiered approach. There are a couple of problems which make this unification tricky. First, P-Store's equations for the cost, time, and effective capacity of the system during reconfiguration must be modified to take skew into account. Second, the scheduling algorithm for reconfigurations will likely need to be modified to offload hot spots quickly while still moving cold data according to P-Store's scheduling algorithm.

Although Squall was built separately from E-Store and P-Store, both elastic systems rely on Squall heavily and have made several modifications to improve its performance. There are many other potential improvements to Squall that should be implemented in future work. For example, clients should receive a copy of each new partition plan so they

send transactions to the correct server. Squall should take better advantage of speculative execution [52] to execute transactions during data migration. To improve its usability, Squall should automatically self-tune in order to find the optimal chunk size and delay between chunks. Future work should also investigate whether other design alternatives to Squall are better suited for moving large amounts of data with high throughput workloads.

Along these lines, it would be interesting to try the elasticity techniques discussed in this thesis on a different DBMS with a different live migration system. VoltDB [101] is an obvious choice since it was inspired by H-Store and shares many of the same design features, but the techniques should apply to other systems as well. B2W uses Riak [56] for their production cart and checkout databases, so building P-Store into Riak would enable B2W to quickly take advantage of this research to help their business.

Another direction is to extend the frameworks to support more complex workloads and applications that have many multi-partition transactions. Clay [83] has already examined this problem for reactive elasticity, but it would be interesting to see if Clay's techniques can be combined with predictive modeling to scale these complex workloads proactively.

Another direction specific to E-Store is developing techniques to reduce the overhead of E-Monitor for more complex workloads. One possible approach is to use approximate frequent item counting algorithms such as SpaceSaving [66] or LossyCount [64].

Specific to P-Store, it would be interesting to try P-Store's techniques on other benchmarks, including those with a less predictable workload, with higher levels of skew, and with some distributed transactions. The performance should decay gracefully as conditions become less ideal.

It would also be useful to make P-Store's reactive response to unexpected load changes configurable. By default, P-Store reacts by scaling out at the same rate R as it uses for proactive reconfigurations. Users should be able to specify if they want to scale out by some multiple of R during reactive reconfigurations.

Finally, it would be interesting to extend the planning algorithms for both systems to take memory consumption into account so that reconfiguration never places more data on a node than its memory capacity. In addition, the monitoring components can be extended to monitor memory usage and trigger reconfiguration if memory on some node is about to

be exhausted.

6.2 Replication vs. Partitioning

E-Store and P-Store are both designed to work with partitioned databases such as H-Store, in which tables are split into disjoint sets of tuples that are distributed across the database cluster. H-Store supports replicating small, read-only tables across all nodes, but currently does not support partial replication of larger tables. The H-Store environment has allowed us to focus exclusively on elasticity, without the added complexity of replication. Future research should examine how to combine elasticity techniques with replication since many existing OLTP DBMSs replicate data for fault tolerance and high availability. These existing systems typically have three replicas of each tuple, but some have as many as six [99] in order to survive disasters such as loss of a data center.

In addition to providing fault tolerance, replicas also impact the performance of the system. An interesting line of research would be to study how different characteristics of the workload impact the optimal number of replicas for each tuple. If a workload is write-heavy, it will likely be best to limit the number of replicas to the minimum number required for fault tolerance. If a workload is read-heavy, however, there may be a benefit to increasing the number of replicas for some tuples. Very hot, read-mostly tuples should likely be replicated many times.

An elastic system could take advantage of replication to not only achieve fault tolerance, but also improve performance. It would need to monitor its workload and dynamically adjust the number of replicas for each tuple in the database to the optimal number given the read/write characteristics and level of skew. These techniques could be combined with E-Store's and P-Store's techniques for dynamic partitioning to build a highly fault tolerant and performant system.

6.3 Scale-up vs. Scale-out

E-Store and P-Store both assume a fixed capacity per node and per partition. Therefore, they achieve elasticity by "scaling out", i.e., adding some number of identical commodity servers to the database cluster. This fits with H-Store's model of one single-threaded execution engine per partition and a fixed number of partitions per node. It would be interesting to examine how elasticity models might change in a different environment in which partitions can be accessed by a variable number of threads, and/or servers can host a variable number of partitions. This would enable elastic DBMSs to "scale up" by keeping the number of servers the same, but increasing the size of each server (e.g., by adding additional DRAM or CPU cores). Public cloud vendors like Amazon AWS have made it easy to scale up by simply choosing a different instance type [5].

When deciding whether to scale up or scale out, an elastic system will likely need to consider several different aspects of the workload. In particular, if a workload has a "tree schema" with many root tuples, it is a good candidate for scaling out. If the workload has many multi-partition transactions, however, it may be a better candidate for scaling up.

Another important consideration will likely be the cost of scaling up. As of August 2017, the cheapest Amazon EC2 instances cost as little as $34 per year, while the most expensive instances cost over $80,000 [5]. Unless the workload is difficult to partition, scaling out will likely be a more cost-effective solution.

6.4 Beyond OLTP and Elasticity

Beyond OLTP databases, there has been a great deal of work on making on-line analytical processing (OLAP) and hybrid transactional/analytical processing (HTAP) databases elastic and adaptable [22, 75, 71, 38, 8]. Future work should investigate whether ideas from OLTP elasticity can be applied to HTAP and OLAP, and vice-versa. Furthermore, it would be interesting to see if techniques for managing shared nothing DBMSs can be applied to the shared storage model, which is becoming increasingly prevalent in cloud-based data warehouses [22, 67]. For example, the elasticity ideas from this thesis could be applied to

the computation layer of a shared storage DBMS, enabling strong cache locality and cache consistency among the compute nodes as the computation layer scales.

Beyond elasticity, there are several other ways to make databases adaptable and self-tuning. Prior work has already investigated automatic index creation [41], automatic materialized view selection and maintenance [63, 106, 70], and automatic adjustment of the storage layout [43, 50, 2, 8]. Future work should investigate other forms of DBMS adaptation and try to combine complimentary techniques into a single database. Combining techniques in this way enables reuse of monitoring features, since many of the approaches require similar statistics about the DBMS. This future study should also attempt to understand the performance overhead caused by adding each additional feature, and quantify the tradeoff between performance and adaptability.

6.5 Summary

There are many interesting questions ripe for investigation in the area of database elasticity. The systems presented in this thesis have the potential for several improvements, but there are other models of elasticity and adaptability worth investigating as well. This chapter presented replication and scaling up as two alternative approaches to elasticity, but there may be many others.

Chapter 7

Conclusion

OLTP applications require high availability and performance from their DBMS, but the workload variability and skew present in many OLTP workloads make it difficult for the DBMS to meet performance requirements in a cost-effective manner. Currently, many companies manage workload variability by provisioning a large cluster of database servers with sufficient capacity to serve the peak workload. This strategy wastes a huge amount of money, power, and hardware since computing resources are underutilized most of the time. It also does not guarantee good performance, since skew may cause one server to become overloaded while others are idle, or a large workload spike may surpass the previous peak and overload the entire cluster.

To enable an OLTP DBMS to use resources efficiently *and* achieve good performance in the presence of workload variability and skew, this thesis has presented two elastic database systems, called E-Store and P-Store. Both systems continuously monitor the load on the DBMS, and use the load statistics collected to determine when and how to reconfigure the database to meet performance requirements and minimize cost. The systems perform reconfiguration without manual intervention, while keeping the database live and maintaining transactional ACID guarantees.

E-Store is designed to maintain system performance over a highly variable and diverse load. It accomplishes this goal by balancing tuple accesses across an elastic set of partitions. The framework consists of two sub-systems, E-Monitor and E-Planner. E-Monitor identifies load imbalances requiring migration based on CPU utilization, and tracks for a

short time window the most-read or -written "hot" tuples. E-Planner chooses which data to move and where to place it. To make intelligent decisions on how to balance the workload across a distributed OLTP DBMS, E-Planner uses smart heuristics. It generates the reconfiguration plan in milliseconds, and the result is a load-balanced system. Moreover, E-Store allows OLTP DBMSs to scale out or in efficiently. The experiments presented in Chapter 3 show that E-Store can start reconfiguring the database after approximately 10 seconds of detecting load skew or a load spike. Reconfiguration results in increasing throughput by up to 4× while reducing latency by up to 10×.

P-Store is a novel database system that uses predictive modeling to elastically reconfigure the database *before* load spikes occur. Chapter 4 defined the problem that P-Store seeks to solve: how to reduce costs by deciding when and how to reconfigure the database. P-Store solves this problem with a novel dynamic programming algorithm for scheduling reconfigurations, as well as a new analytical model for shared nothing reconfiguration and parallel migration. To accurately predict the load for different applications, P-Store uses a time-series model called Sparse Periodic Auto-Regression (SPAR) [17]. Chapter 4 shows an evaluation of running a real online retail workload in H-Store and using P-Store's predictive models to decide when and how to reconfigure, thus demonstrating the cost savings that can be achieved with P-Store.

Although the ideas presented in this thesis are in the context of two distinct systems, the techniques are complementary, and future work should unify the ideas in a single system. Furthermore, these ideas are in the context of one model of elasticity based on repartitioning and scaling out. Future work should investigate whether other models can further improve performance for some workloads. By answering these questions and continuing to investigate techniques for database elasticity and adaptability, we can ensure that future OLTP DBMSs will be cost efficient, high performance, and fully autonomous.

Appendix A

Symbols Used Throughout Thesis

For ease of reference, Tables A.1 to A.3 list the symbols used throughout the thesis in the order they appear.

Symbol	Definition
W	Length of the monitoring time window
$\{r_1, r_2, \ldots, r_m\}$	Set of all tuples (records) in the database
$\{p_1, p_2, \ldots, p_c\}$	Set of partitions
$L(r_i)$	The load (access count) on tuple r_i
$L(p_j)$	Sum of tuple accesses for partition p_j
$TK(p_j)$	Set of the top-k most frequently accessed tuples for partition p_j
B	Size of each block of cold tuples
A	Average load per partition
$A + \varepsilon$	Maximum load allowed per partition after bin packing
$x_{i,j} \in \{0,1\}$	Binary decision variable in bin packing algorithm for assignment of hot tuple r_i to partition p_j
$y_{k,j} \in \{0,1\}$	Binary decision variable in bin packing algorithm for assignment of cold block b_k to partition p_j
n	Number of hot tuples in the database
d	Number of cold tuple blocks in the database
c	Number of partitions
T	Transmission cost of moving a tuple
$t_{i,j} \in \{0,T\}$	Transmission cost of assigning tuple r_i to partition p_j

Table A.1: Symbols and definitions used in Chapter 3

Symbol	Definition
C	Cost of a DBMS cluster over T time intervals
T	Number of time intervals considered in calculation of C
s_t	Number of servers in the database cluster at time t
Q	The target average throughput of a single database server
$\hat{Q}$	The maximum throughput of a single database server
D	The time needed to move all data in the database once with a single thread
R	The rate at which data must be migrated to move the entire database in time D
B	Number of servers before a reconfiguration
A	Number of servers after a reconfiguration
move	A reconfiguration from A to B servers
L	Time-series array of predicted load of of length T
N_0	Number of nodes allocated at the start of Algorithm 1
Z	The maximum number of machines needed to serve the predicted load in L
m	A matrix to memoize the cost and best series of moves calculated by Algorithm 2
M	The sequence of moves returned by Algorithm 1
`cap`(N)	Returns the maximum capacity of N servers
$T(B,A)$	Returns the time for a reconfiguration from B to A servers
$C(B,A)$	Returns the cost of a reconfiguration from B to A servers
`eff-cap`(B,A,f)	Returns the effective capacity of the DBMS after fraction f of the data has been moved when reconfiguring from B to A servers
P	The number of partitions per server
$max_{\parallel}$	The maximum number of parallel migrations during a reconfiguration

Table A.2: Symbols and definitions used in Chapter 4

Symbol	Definition
`avg-mach-alloc`(B,A)	Returns the average number of machines allocated when reconfiguring from B to A servers
s	Minimum of B and A
l	Maximum of B and A
Δ	The difference between s and l
r	The remainder of dividing Δ by s
f_n	The fraction of the database hosted by node n
f	The fraction moved so far of the total data moving during a reconfiguration
τ	The SPAR forecasting window
a_k	SPAR coefficient for periodic load
b_j	SPAR coefficient for recent load
$y(t+\tau)$	SPAR forecasted load at time $t+\tau$
n	The number of previous periods considered by SPAR
m	The number of recent load measurements considered by SPAR

Table A.3: Symbols and definitions used in Chapter 4, continued

Bibliography

[1] Sherif Akoush, Ripduman Sohan, Andrew Rice, Andrew W Moore, and Andy Hopper. Predicting the performance of virtual machine migration. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2010 IEEE International Symposium on*, pages 37–46. IEEE, 2010.

[2] Ioannis Alagiannis, Stratos Idreos, and Anastasia Ailamaki. H2O: A Hands-free Adaptive Store. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 1103–1114, New York, NY, USA, 2014. ACM.

[3] Amazon AWS Startups. `https://aws.amazon.com/startups/`. [Online; accessed: 01-Aug-2017].

[4] Amazon DynamoDB. `https://aws.amazon.com/dynamodb/faqs/`. [Online; accessed: 28-Jul-2017].

[5] Amazon EC2 Pricing. `https://aws.amazon.com/ec2/dedicated-hosts/pricing/`. [Online; accessed: 19-Mar-2017].

[6] Ioannis Arapakis, Xiao Bai, and B Barla Cambazoglu. Impact of response latency on user behavior in web search. In *Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval*, pages 103–112. ACM, 2014.

[7] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A View of Cloud Computing. *Commun. ACM*, 53(4):50–58, April 2010.

[8] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. Bridging the Archipelago Between Row-Stores and Column-Stores for Hybrid Workloads. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 583–598, New York, NY, USA, 2016. ACM.

[9] Matthew Aslett. How will the database incumbents respond to NoSQL and NewSQL. *San Francisco, The*, 451:1–5, 2011.

[10] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer networks*, 54(15):2787–2805, 2010.

[11] B2W Digital. https://www.b2wdigital.com, 2017.

[12] David F. Bacon, Nathan Bales, Nico Bruno, Brian F. Cooper, Adam Dickinson, Andrew Fikes, Campbell Fraser, Andrey Gubarev, Milind Joshi, Eugene Kogan, Alexander Lloyd, Sergey Melnik, Rajesh Rao, David Shue, Christopher Taylor, Marcel van der Holst, and Dale Woodford. Spanner: Becoming a SQL System. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 331–343, New York, NY, USA, 2017. ACM.

[13] Sean Kenneth Barker, Yun Chi, Hakan Hacigümüs, Prashant J Shenoy, and Emmanuel Cecchet. ShuttleDB: Database-Aware Elasticity in the Cloud. In *ICAC*, pages 33–43, 2014.

[14] Sean Kenneth Barker, Yun Chi, Hyun Jin Moon, Hakan Hacigümüş, and Prashant J. Shenoy. "Cut me some slack": latency-aware live migration for databases. In *EDBT*, 2012.

[15] Deborah Barnes and Vijay Mookerjee. Customer delay in e-Commerce sites: Design and strategic implications. *Business Computing*, 3:117, 2009.

[16] Jake Brutlag. Speed Matters for Google Web Search. https://services.google.com/fh/files/blogs/google_delayexp.pdf, 2009. [Online; accessed: 16-Mar-2017].

[17] Gong Chen, Wenbo He, Jie Liu, Suman Nath, Leonidas Rigas, Lin Xiao, and Feng Zhao. Energy-aware Server Provisioning and Load Dispatching for Connection-intensive Internet Services. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 337–350, Berkeley, CA, USA, 2008. USENIX Association.

[18] Kristina Chodorow and Michael Dirolf. *MongoDB: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2010.

[19] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, 2010.

[20] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. Schism: A Workload-driven Approach to Database Replication and Partitioning. *PVLDB*, 3(1-2), 2010.

[21] Carlo Curino, Evan P. C. Jones, Samuel Madden, and Hari Balakrishnan. Workload-aware database monitoring and consolidation. In *SIGMOD*, 2011.

[22] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. The

Snowflake Elastic Data Warehouse. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 215–226, New York, NY, USA, 2016. ACM.

[23] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. ElasTraS: An Elastic, Scalable, and Self-Managing Transactional Database for the Cloud. *ACM Transactions on Database Systems*, 38(1):5:1–5:45, 2013.

[24] Sudipto Das, Shoji Nishimura, Divyakant Agrawal, and Amr El Abbadi. Albatross: Lightweight Elasticity in Shared Storage Databases for the Cloud Using Live Data Migration. *PVLDB*, 4(8), 2011.

[25] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[26] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: Sql server's memory-optimized oltp engine. In *Proceedings of the 2013 international conference on Management of data*, pages 1243–1254. ACM, 2013.

[27] Jennie Duggan and Michael Stonebraker. Incremental Elasticity for Array Databases. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 409–420, New York, NY, USA, 2014. ACM.

[28] Aaron J. Elmore, Vaibhav Arora, Rebecca Taft, Andrew Pavlo, Divyakant Agrawal, and Amr El Abbadi. Squall: Fine-Grained Live Reconfiguration for Partitioned Main Memory Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 299–313, New York, NY, USA, 2015. ACM.

[29] Aaron J. Elmore, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. Towards an Elastic and Autonomic Multitenant Database. In *NetDB*, 2011.

[30] Aaron J. Elmore, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. Zephyr: Live Migration in Shared Nothing Databases for Elastic Cloud Platforms. In *SIGMOD*, 2011.

[31] Ilir Fetai, Damian Murezzan, and Heiko Schuldt. Workload-driven adaptive data partitioning and distribution - The Cumulus approach. In *Big Data (Big Data), 2015 IEEE International Conference on*, pages 1688–1697. IEEE, 2015.

[32] VNI Global Fixed. Mobile Internet Traffic Forecasts, 2016.

[33] Nathan Folkman. So, that was a bummer. `http://is.gd/SRF0sb`, 2010. [Online; accessed: 20-Aug-2017].

[34] Wikimedia Foundation. Wikipedia page view statistics. https://dumps.wikimedia.org/other/pagecounts-raw, 2017. [Online; accessed: 20-Aug-2017].

[35] Prasanna Ganesan, Mayank Bawa, and Hector Garcia-Molina. Online balancing of range-partitioned data with applications to peer-to-peer systems. In *VLDB*, 2004.

[36] Jonathan Gaw. Heavy Traffic Crashes Britannica's Web Site – Los Angeles Times. http://lat.ms/1fXLjYx, 1999. [Online; accessed: 20-Aug-2017].

[37] D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper. Workload Analysis and Demand Prediction of Enterprise Data Center Applications. In *2007 IEEE 10th International Symposium on Workload Characterization*, pages 171–180, Sept 2007.

[38] Anil K. Goel, Jeffrey Pound, Nathan Auch, Peter Bumbulis, Scott MacLean, Franz Färber, Francis Gropengiesser, Christian Mathis, Thomas Bodner, and Wolfgang Lehner. Towards Scalable Real-time Analytics: An Architecture for Scale-out of OLxP Workloads. *Proc. VLDB Endow.*, 8(12):1716–1727, August 2015.

[39] Zhenhuan Gong, Xiaohui Gu, and John Wilkes. PRESS: PRedictive Elastic ReSource Scaling for cloud systems. In *Network and Service Management (CNSM), 2010 International Conference on*, pages 9–16. IEEE, 2010.

[40] Google Cloud Spanner. https://cloud.google.com/spanner/. [Online; accessed: 21-Aug-2017].

[41] Goetz Graefe, Felix Halim, Stratos Idreos, Harumi Kuno, Stefan Manegold, and Bernhard Seeger. Transactional support for adaptive indexing. *The VLDB Journal*, 23(2):303–328, 2014.

[42] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1992.

[43] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudre-Mauroux, and Samuel Madden. HYRISE: A Main Memory Hybrid Storage Engine. *Proc. VLDB Endow.*, 4(2):105–116, November 2010.

[44] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. OLTP Through the Looking Glass, and What We Found There. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 981–992, New York, NY, USA, 2008. ACM.

[45] V Holub. Java implementation of MurmurHash. https://github.com/tnm/murmurhash-java, 2010. [Online; accessed: 29-Mar-2017].

[46] M. Holze, A. Haschimi, and N. Ritter. Towards workload-aware self-management: Predicting significant workload shifts. In *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*, pages 111–116, March 2010.

[47] Yu-Ju Hong and Mithuna Thottethodi. Understanding and Mitigating the Impact of Load Imbalance in the Memory Caching Tier. In *SoCC*, 2013.

[48] H-Store: A Next Generation OLTP DBMS. http://hstore.cs.brown.edu. [Online; accessed: 21-Aug-2017].

[49] Daniel Jacobson, Danny Yuan, and Neeraj Joshi. Scryer: Netflix's Predictive Auto Scaling Engine. http://techblog.netflix.com/2013/11/scryer-netflixs-predictive-auto-scaling.html, 2013. [Online; accessed: 1-Apr-2017].

[50] Alekh Jindal and Jens Dittrich. Relax and let the database do the partitioning online. In *International Workshop on Business Intelligence for the Real-Time Enterprise*, pages 65–80. Springer, 2011.

[51] Evan P.C. Jones. *Fault-Tolerant Distributed Transactions for Partitioned OLTP Databases*. PhD thesis, MIT, 2012.

[52] Evan P.C. Jones, Daniel J. Abadi, and Samuel Madden. Low Overhead Concurrency Control for Partitioned Main Memory Databases. In *SIGMOD '10: Proceedings of the 2010 international conference on Management of data*, pages 603–614, New York, NY, USA, 2010. ACM.

[53] David Josephsen. *Building a Monitoring Infrastructure with Nagios*. Prentice Hall PTR, USA, 2007.

[54] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-Store: A High-performance, Distributed Main Memory Transaction Processing System. *PVLDB*, 1(2), 2008.

[55] Kissmetrics. How Loading Time Affects Your Bottom Line. https://blog.kissmetrics.com/loading-time/?wide=1, 2017. [Online; accessed: 28-Feb-2017].

[56] Rusty Klophaus. Riak core: Building distributed applications without shared state. In *ACM SIGPLAN Commercial Users of Functional Programming*, page 14. ACM, 2010.

[57] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[58] Greg Linden. Make Data Useful. http://www.gduchamp.com/media/StanfordDataMining.2006-11-28.pdf, 2006. [Online; accessed: 28-Feb-2017].

[59] Greg Linden. Marissa Mayer at Web 2.0. http://glinden.blogspot.com/2006/11/marissa-mayer-at-web-20.html, 2006. [Online; accessed: 16-Mar-2017].

[60] Haikun Liu, Hai Jin, Cheng-Zhong Xu, and Xiaofei Liao. Performance and energy modeling for live migration of virtual machines. *Cluster Computing*, 16(2):249–264, 2013.

[61] Zhenhua Liu, Yuan Chen, Cullen Bash, Adam Wierman, Daniel Gmach, Zhikui Wang, Manish Marwah, and Chris Hyser. Renewable and Cooling Aware Workload Management for Sustainable Data Centers. *SIGMETRICS Perform. Eval. Rev.*, 40(1):175–186, June 2012.

[62] Nirmesh Malviya, Ariel Weisberg, Samuel Madden, and Michael Stonebraker. Rethinking main memory OLTP recovery. In *ICDE*, 2014.

[63] Imene Mami and Zohra Bellahsene. A survey of view selection methods. *SIGMOD Rec.*, 41(1):20–29, April 2012.

[64] Gurmeet Singh Manku and Rajeev Motwani. Approximate frequency counts over data streams. In *VLDB*, 2002.

[65] David Q Mayne and Hannah Michalska. Receding horizon control of nonlinear systems. *IEEE Transactions on automatic control*, 35(7):814–824, 1990.

[66] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *ICDT*, 2005.

[67] Microsoft Azure SQL DW. `https://azure.microsoft.com/en-us/services/sql-data-warehouse/`.

[68] Umar Farooq Minhas, Rui Liu, Ashraf Aboulnaga, Kenneth Salem, Jonathan Ng, and Sean Robertson. Elastic scale-out for partition-based database systems. In *ICDE Workshops*, 2012.

[69] Bob Minzesheimer. How the 'Oprah Effect' changed publishing. *USA Today*, 22 May 2011. `https://usatoday30.usatoday.com/life/books/news/2011-05-22-Oprah-Winfrey-Book-Club_n.htm`. [Online; accessed: 16-Aug-2017].

[70] Hoshi Mistry, Prasan Roy, S Sudarshan, and Krithi Ramamritham. Materialized view selection and maintenance using multi-query optimization. In *ACM SIGMOD Record*, pages 307–318. ACM, 2001.

[71] Tobias Mühlbauer, Wolf Rödiger, Angelika Reiser, Alfons Kemper, and Thomas Neumann. Scyper: Elastic olap throughput on transactional data. In *Proceedings of the Second Workshop on Data Analytics in the Cloud*, DanaC '13, pages 11–15, New York, NY, USA, 2013. ACM.

[72] Alex Nazaruk and Michael Rauchman. Big data in capital markets. In *ICMD*, 2013.

[73] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In *NSDI*, 2013.

[74] NuoDB. `http://www.nuodb.com`.

[75] Jennifer Ortiz, Brendan Lee, and Magdalena Balazinska. PerfEnforce Demonstration: Data Analytics with Performance Guarantees. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 2141–2144, New York, NY, USA, 2016. ACM.

[76] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *SIGMOD*, 2012.

[77] Andrew Pavlo, Evan P C Jones, and Stan Zdonik. On Predictive Modeling for Optimizing Transaction Execution in Parallel OLTP Systems. *PVLDB*, 5(2):85–96, 2011.

[78] Tilmann Rabl and Hans-Arno Jacobsen. Query centric partitioning and allocation for partially replicated database systems. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 315–330, New York, NY, USA, 2017. ACM.

[79] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, Inc., New York, NY, USA, 3 edition, 2003.

[80] Domenico Sacca and Gio Wiederhold. Database partitioning in a cluster of processors. *ACM Trans. Database Syst.*, 10(1):29–56, March 1985.

[81] Oliver Schiller, Nazario Cipriani, and Bernhard Mitschang. ProRea: Live Database Migration for Multi-Tenant RDBMS with Snapshot Isolation. In *EDBT*, 2013.

[82] Marco Serafini, Essam Mansour, Ashraf Aboulnaga, Kenneth Salem, Taha Rafiq, and Umar Farooq Minhas. Accordion: Elastic scalability for database systems supporting distributed transactions. *PVLDB*, 7(12), 2014.

[83] Marco Serafini, Rebecca Taft, Aaron J Elmore, Andrew Pavlo, Ashraf Aboulnaga, and Michael Stonebraker. Clay: fine-grained adaptive partitioning for general database schemas. *Proceedings of the VLDB Endowment*, 10(4):445–456, 2016.

[84] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. CloudScale: Elastic Resource Scaling for Multi-tenant Cloud Systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SoCC '11, pages 5:1–5:14, New York, NY, USA, 2011. ACM.

[85] Matthew Sladescu. *Proactive Event Aware Cloud Elasticity Control*. PhD thesis, University of Sydney, 2015.

[86] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, 2001.

[87] Radu Stoica, Justin J. Levandoski, and Per-Ake Larson. Identifying hot and cold data in main-memory databases. In *ICDE*, 2013.

[88] Michael Stonebraker. The case for shared nothing. *IEEE Database Eng. Bull.*, 9(1):4–9, 1986.

[89] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (it's time for a complete rewrite). In *VLDB*, 2007.

[90] Michael Stonebraker and Ariel Weisberg. The VoltDB main memory DBMS. *IEEE Data Eng. Bull*, 36(2), 2013.

[91] Rebecca Taft. B2W Benchmark in H-Store. `https://github.com/rytaft/h-store/tree/b2w/src/benchmarks/edu/mit/benchmark/b2w`, 2017.

[92] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. E-Store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proceedings of the VLDB Endowment*, 8(3):245–256, 2014.

[93] Aubrey L. Tatarowicz, Carlo Curino, Evan P. C. Jones, and Sam Madden. Lookup tables: Fine-grained partitioning for distributed databases. In *ICDE*. IEEE Computer Society, 2012.

[94] Selome Kostentinos Tesfatsion, Eddie Wadbro, and Johan Tordsson. Autonomic Resource Management for Optimized Power and Performance in Multi-tenant Clouds. In *Autonomic Computing (ICAC), 2016 IEEE International Conference on*, pages 85–94. IEEE, 2016.

[95] Pınar Tözün, Ippokratis Pandis, Ryan Johnson, and Anastasia Ailamaki. Scalable and dynamically balanced shared-everything oltp with physiological partitioning. *The VLDB Journal*, 22(2):151–175, 2013.

[96] The TPC-C Benchmark, 1992. http://www.tpc.org/tpcc/.

[97] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32. ACM, 2013.

[98] Guido Urdaneta, Guillaume Pierre, and Maarten Van Steen. Wikipedia workload analysis for decentralized hosting. *Computer Networks*, 53(11):1830–1845, 2009.

[99] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1041–1052, New York, NY, USA, 2017. ACM.

[100] Michail Vlachos, Christopher Meek, Zografoula Vagena, and Dimitrios Gunopulos. Identifying similarities, periodicities and bursts for online search queries. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 131–142, New York, NY, USA, 2004. ACM.

[101] VoltDB. `http://www.voltdb.com`.

[102] William Voorsluys, James Broberg, Srikumar Venugopal, and Rajkumar Buyya. Cost of virtual machine live migration in clouds: A performance evaluation. In *IEEE International Conference on Cloud Computing*, pages 254–265. Springer, 2009.

[103] Cheng Wang, Bhuvan Urgaonkar, Aayush Gupta, Lydia Y Chen, Robert Birke, and George Kesidis. Effective capacity modulation as an explicit control knob for public cloud profitability. In *Autonomic Computing (ICAC), 2016 IEEE International Conference on*, pages 95–104. IEEE, 2016.

[104] Craig D. Weissman and Steve Bobrowski. The design of the force.com multitenant internet application development platform. In *SIGMOD*, pages 889–896, 2009.

[105] Fan Yang, Jayavel Shanmugasundaram, and Ramana Yerneni. A scalable data platform for a large number of small applications. *CIDR*, 1(3):11, 2009.

[106] J. Zhou, P. A. Larson, J. Goldstein, and L. Ding. Dynamic materialized views. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 526–535, April 2007.