Abstract of "On Scalable Transaction Execution in Partitioned Main Memory Database Management Systems" by Andrew Pavlo, Ph.D., Brown University, May 2014.

An emerging class of distributed database management systems (DBMS) seek to provide high-performance transaction processing for data-intensive applications while maintaining strong consistency guarantees. These systems achieve this by storing databases in a cluster of shared-nothing, main memory partitions. This partitioning enables them to eschew much of the legacy, disk-oriented architecture that slows down traditional systems, such as heavy-weight concurrency control algorithms, thereby allowing for the efficient execution of single-partition transactions. But many applications cannot be partitioned such that all of their transactions execute in this manner; these multi-partition transactions require expensive coordination that inhibits performance. In this dissertation, we study the problem of scalable transaction execution for modern on-line transaction processing (OLTP) applications. We present the design of H-Store, a distributed, main memory DBMS that is optimized for short-lived, write-heavy transactional workloads. We then present domain-specific applications of optimization and machine learning techniques to improve the performance of fast database systems like H-Store. Our first contribution is an approach for automatically generating a database design for an OLTP application that both minimizes both the amount of cross-node communication and skew in the cluster. We then present a Markov model-based approach for automatically selecting which optimizations to enable at run time for new transaction requests based on their most likely behavior. Finally, we present a method for using these models to schedule speculative transactions and queries whenever a DBMS stalls because of a multi-partition transaction. All together, these allow enable H-Store to support transactional workloads that are beyond what single-node systems can handle.

On Scalable Transaction Execution in Partitioned Main Memory Database Management Systems

by
Andrew Pavlo

A dissertation submitted in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy
in the Department of Computer Science at Brown University

Providence, Rhode Island
May 2014

This dissertation by Andrew Pavlo is accepted in its present form by
the Department of Computer Science as satisfying the dissertation requirement
for the degree of Doctor of Philosophy.

Date _____ _____
Stanley B. Zdonik, Director

Recommended to the Graduate Council

Date _____ _____
Michael Stonebraker, Reader
(Massachusetts Institute of Technology)

Date _____ _____
Ugur Cetintemel, Reader
(Brown University)

Approved by the Graduate Council

Date _____ _____
Peter Weber
Dean of the Graduate School

♡ K.B.

Acknowledgements

The best part of my time in graduate school was the chance to work with Stan Zdonik at Brown. I cannot thank him enough for his guidance and patience with me after the various "misunderstandings" about me in the Brown Computer Science department. I am grateful that he always had time to discuss research with me, but that he also gave me the freedom to pursue my own research agenda. After VoltDB forked the H-Store source code in 2008, Stan pushed me to continue developing the original system rather than relying on the commercial version. This was not an easy choice and Stan stood by me as I spent almost two years building it out so that we could run the experiments in this dissertation. In hind sight, this was the correct choice and I attribute much of my success to this decision. I could not have asked for a better advisor.

I am fortunate for having the opportunity to work with Mike Stonebraker and Sam Madden at MIT. Mike's years of experience in database systems were helpful for me to figure out what were interesting research problems and how to distill issues down to their simplest form. I also greatly enjoyed working with Sam because of his ability to quickly understand problems and then provide useful feedback or incisive questions during meetings (even though he may seem aloof at times).

I would like to also thank the original members of the H-Store team: Evan Jones (MIT), Hideaki Kimura (Brown), Jonathan Natkins (Brown), Yang Zhang (MIT), and John Hugg (VoltDB). Evan was particularly helpful later on when figuring out how to leverage the capabilities of our prediction framework (cf. Chapter 5) in H-Store. I later worked on the automatic partitioning component (cf. Chapter 4) in H-Store with Carlo Curino who was helpful in getting the paper published after multiple rejections. At Brown, I had help in writing benchmark and utility code from Visawee Angkanawaraphan, Zhe Zhang, and Xin Jia. I am also appreciative of the fruitful conversations about machine learning and optimization problems that I had with Yuri Malitsky and Carleton Coffrin at Brown. Finally, I would also like to thank Leon Wrinkles for his invaluable (albeit strident) criticisms and suggestions on the speculative execution project (cf. Chapter 6).

There are many others that I have worked with on other research projects that are not part of this dissertation. Foremost is Ning Shi, who stood by me after we got in trouble for the Graffiti Network project. David DeWitt at Microsoft, Eric Paulson at the University of Wisconsin–Madison, Dan Abadi at Yale, and Alex Rasin at Brown for their help on the MapReduce analysis paper. Yang Zou and Emanuel Buzek at Brown for their help with the MongoDB designer project. Djellel Difallah and Philippe Cudre-Mauroux at the University of Fribourg for their help on the OLTP-Bench framework.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Changes in Internet usage trends in the last decade have given rise to numerous Web-based, front-end applications that support a large number of concurrent users. Creating one of these large-scale, data-intensive applications is easier now than it ever has been, in part due to the proliferation of open-source distributed system tools, cloud-computing platforms, and affordable mobile sensors. Developers are able to deploy applications in a short amount of time that have the potential to reach millions of users and collect large amounts of data from a variety of sources.

These large-scale database applications are generally divided into two parts: (1) the front-end transaction processing system and (2) the back-end data warehouse. A *transaction processing system* is a computer system that is designed to execute *on-line transaction processing* (OLTP) workload of the application [32, 95]. New data and updates from the transaction processing system are then streamed or bulk loaded to the back-end data warehouse. This is where an organization can execute an *on-line analytical processing* (OLAP) workloads for extrapolating information from the entire database.

The term transaction processing system is generally used to mean a complete system that includes application, operations tools, one or more database management systems (DBMSs), utilities, and networking and operating system software [95]. On one end of such a system there is a *front-end application* that contains the code that interfaces with the "outside" world. This is the part of the system that interacts with *end-users*; for example, it is the system that processes new orders, responds to a page request, or performs a financial transaction. Such users could be humans on their personal computer or mobile device, or another computer program potentially running somewhere else in the world.

One of the first OLTP applications was the SABRE airline reservation system developed in the 1960s [32]. It was one of the biggest computer systems built at the time and is still used today by over 200 airlines and thousands of travel agencies. Other OLTP workloads include maintaining the state of Internet games, real-time advertising on web pages, and processing incoming message streams (either from humans or machines) in order to maintain some runtime state.

End-users interact with the front-end application by sending it requests to perform some function (e.g., reserve a seat on a flight). The application processes these requests and then executes transactions in the DBMS. A *transaction* in the context of one of these systems is the execution of a sequence of one or more

operations (e.g., SQL queries) on a shared database to perform some higher-level function [32, 94, 95]. It is the basic unit of change in a DBMS: partial transactions are not allowed, and the effect of a group of transactions on the database's state is equivalent to any serial execution of all transactions [185].

These transactions invoke operations that either read or modify the current state of the database, and then either commit or abort. If the transaction commits, then the DBMS saves any changes that it made to the database and then returns its result to the application. If it aborts, then the modifications that the transactions made to the database are reverted.

A transaction processing system is expected to maintain four properties for each transaction that it executes: (1) atomicity, (2) consistency, (3) isolation, and (4) durability [95, 185]. These unifying concepts for distributed computation are collectively referred to with the *ACID* acronym [106]:

Atomicity: The changes that a transaction makes to the state of the database either all happen or none of them happen at all. If a transaction aborts, then the state of the database will be as if the transaction never executed at all. There are no partial changes.

Consistency: A transaction performs a correct transformation of the database's state. The transaction's operations taken as a group do not violate any of the integrity constraints associated with the state [95]. Examples of a constraint are that all of the primary key values are unique or that all foreign key references are valid [32]. It is assumed that the program logic contained within the front-end application itself is correct [177].

Isolation: This means that even though the system may execute transactions concurrently, it appears to each transaction that all other transactions executed either before it or after it. A system can provide different levels of isolation. The strongest isolation level, known as *serializability*, means that each transaction executes as if it has exclusive access to the database and no transaction can see the changes made by another incomplete transaction [32].

Durability: The changes made to the database by a completed transaction are persistent (i.e., they will survive failures and system restarts). If the application receives an acknowledgement that a transaction has committed, then at any point in the future the application should be able to retrieve the updates made to the database by that transaction (assuming that they were not overwritten by another transaction).

Transactions are a useful abstraction for developers as it allows them to reason more easily about concurrent programs. The ACID guarantees of transactions are essential in any application that deals with high-profile information, where an inconsistent or lost operation may result in a significant financial loss to a company. Achieving good performance in a transaction processing system that supports ACID transactions is notoriously difficult [127, 143, 211], but it is believed that it is better to have application programmers deal with performance problems due to overuse of transactions, rather than always coding around the lack of transactions [63].

Many of the DBMSs used in these front-end transaction processing systems are based on the "traditional" system architecture that was developed in the 1970s. This architecture comes from the pioneering work on the original relational DBMSs, IBM's System R [24] and the University of California's INGRES [209]. These

early DBMSs were designed for the computers that were available at that time; a typical computer had a single CPU core and a small amount of main memory. They were also developed at a time when the primary market for databases was interactive transaction processing applications. Most commercial DBMSs in today's market are based on this traditional architecture model, including Oracle, IBM's DB2, and Microsoft's SQL Server, as well as popular open-source DBMSs, such as Postgres and MySQL.

But nearly four decades after System R and INGRES were created, the processing and storage needs of modern OLTP applications are now surpassing the abilities of these legacy systems [11, 87, 198, 201]. With the number of Internet-enabled devices currently estimated to be in the billions [12], even a medium-sized Internet application can become popular enough that it will need to service thousands of requests per second. Larger transaction processing systems can have workloads with millions of active users at the same time. A traditional DBMS deployed on a single node is unable to keep up with these demanding workloads.

To better understand why existing database systems are insufficient for modern transaction processing applications, we now discuss the key characteristics of OLTP workloads that make high-performance transaction processing particularly challenging. We then discuss how traditional DBMS architectures are inadequate for modern processor technologies and memory capacities.

1.1 Modern On-Line Transaction Processing Workloads

The workloads for today's OLTP applications are much different than the business processing applications from the 1970s. In these earlier applications, a human operator started a transaction through a terminal and then entered new data into the system manually. Back then, the expected peak throughput of a DBMSs was only tens to hundreds transactions per second [83, 212] and the response times were measured in seconds [96].

Once business and organizations collected a large amount of data from these front-end systems, they then wanted to analyze it to extrapolate new information to guide their decision making. Thus, during the 1990s, DBMS vendors extended traditional DBMSs to include support for more complex OLAP workloads [13, 207, 211]. But modern OLTP workloads are distinct from OLAP workloads, and thus these enhancements do not help the DBMS to support high-performance transaction processing applications.

In general, we characterize transactions in OLTP applications as having the following four properties:

Small Footprint: Each transaction only accesses a small subset of the entire database. A transaction uses indexes to retrieve a small number of tuples and only performs its operations on that data. These look-ups are based on a key provided by the application that corresponds to a single "entity" in the database (e.g., the account number for a single customer). This key is also used to retrieve or update the records related to that entity (e.g., the order records for a single customer). OLTP transactions do not perform full table scans or large distributed joins. This differs from OLAP systems, where queries tend to compute aggregate information from multiple entities in the database (e.g., the average order amount for all customers) [13, 210]

Short-lived: As a result of having a small footprint in the database, each transaction only runs for a short amount of time. Their response time requirements are measured in milliseconds, rather minutes or seconds. In contrast, OLAP queries tend to run for longer periods of time because they have to read

more data [13]. The short execution times for OLTP transactions also imply that they never stall waiting for additional input from an operator; all of the input that the transaction needs to complete is provided in the transaction request.

Write-Heavy: The workloads for OLTP applications are inherently write-heavy because the end-user is performing some action that causes the application to update its state. Most transactions update the state of the database by inserting new records or updating existing ones. Again, because each transaction has a small footprint, new data is added to the database in small batches. This differs from OLAP systems, where the workload is primarily read-only and new data is bulk loaded.

Repetitive: End-users have fixed number of ways that they can interact with the application (e.g., through a web site). Thus, in an OLTP application, there are a limited number of transaction types. Each transaction type executes the same set of pre-defined, parameterized queries whose variables are filled in at runtime. The workloads for OLAP systems, on the other hand, are less predictable. This is common in exploratory data analysis applications, where the user often does not know what they are looking for in the database right away and will execute random queries.

OLTP workloads can also contain ad hoc (or "one-shot") queries that are interspersed with the regular transactional workload. These queries are used to make rare, one time modifications to the database (e.g., apply a discount to an existing order for a customer due to a mistake). These queries are still short-lived and only touch a small amount of data. Other one-shot operations include analytical queries to compute a simple metric for a particular aspect of the database (e.g., count the number of items sold within the last hour). These types of queries are also rare in an OLTP workload and are customarily executed on a separate OLAP or data warehouse DBMS [210].

1.2 Modern Computer Hardware

Current hardware trends are much different now than they were in the 1970s. Back then, a DBMS was usually deployed on a computer that only had a single CPU with a single core (i.e., hardware thread). The amount of memory available on these machines was much smaller relative to the size of the database back then. Thus, a traditional DBMS assumes that most of the database is stored on disk. But with the advent of large-memory computers, it is now affordable to purchase a small number of machines that have almost a terabyte of DRAM. This is enough main memory to store all but the largest OLTP databases [211]. In our experience, the databases' for modern OLTP applications are typically several hundred gigabytes in size. The largest one that we are aware of is approximately 10 terabytes. Contrast this with an OLAP data warehouse, where the DBMS could be managing databases that are several petabytes in size. This difference is because an OLTP database stores the current state of an application (e.g., the orders from the last 90 days), whereas an OLAP database stores all of the historical information for an enterprise (e.g., all orders ever placed).

Multi-core processors also a challenge for traditional DBMSs. Earlier systems supported concurrent transactions [52], but since the CPUs at the time only had a single hardware thread that meant that only one transaction would actually be running at a time. Thus, these DBMSs employed a lock-based concurrency control schemes to ensure correctness when one transaction's thread would block while another thread

started running [98]. But now with multiple cores, there could be transactions running at the same time in the DBMS on a single computer. The concurrency control scheme still ensures correctness, but now lock contention impedes performance [127, 211]. Hence, adding additional cores does not automatically improve the performance of a traditional DBMS. This is problematic because manufacturers are not creating chips with higher clock speeds due to heat and energy limitations, and instead are adding more cores to a single CPU.

1.3 Implications for DBMS Architectures

There is substantial evidence that shows that many organizations struggle with scaling traditional DBMSs for modern OLTP applications [198, 201]. These difficulties strongly demonstrate the need to rethink system architectures for high-performance transaction processing. Rather than improve on the existing design decisions of traditional DBMS architectures, we contend that a new "clean-slate" architecture that is specifically designed for the high throughput and low latency requirements of modern OLTP applications is a better approach.

It has long been observed that a main memory DBMS outperforms a disk-oriented systems [72, 88, 145]. If a database is stored in memory, rather than disk, then the concurrency control techniques employed by traditional DBMSs to mask disk access latency and provide the illusion of the serial execution of multiple transactions will take just as long to execute as accessing the data itself. That is, if the data needed by a transaction is already in memory, then it takes just as long to access that data as it does for the DBMS's concurrency control scheme to set a lock or latch in memory.

But the amount of memory available on a single node might not be enough for some applications. Furthermore, if the database is only stored on a single node, then it may take a long time to get an OLTP application back on-line if that node fails. All of this argues for the use of a distributed DBMS architecture [175] where the database is deployed on a cluster of shared-nothing nodes [206]. In addition to this, since an OLTP application is what end-users typically interact with, it is important that the system is always on-line and available. The cost of an outage for these applications is significant [22], thus the DBMS must be tolerant to failures and reduce the amount of time that it is off-line because of them.

Although distributed DBMSs are a well-studied research area [70, 160], much of the previous work assumes that the DBMS operates on single-CPU, disk-based machines [73]. The demands of modern enterprise transactional and Web-based workloads, as well as changes in infrastructure and CPU architecture trends, means that many of the design decisions and techniques used in previous systems must be re-examined. For example, just because a database is now stored in memory across multiple nodes does not mean that it will automatically perform well. Moving to a multi-node architecture introduces a new performance bottleneck: *distributed transactions* [30, 173]. These transactions access data that is stored at two or more nodes in the database cluster. Since transactions may modify data at those nodes, a distributed DBMS must use an atomic commit protocol, such as *two-phase commit* (2PC) [35, 93], to ensure that operations occur in the right order and that the state of the database is correct across all nodes. The coordination overhead of such protocols inhibits the scalability of distributed DBMSs, because the network becomes the main bottleneck [113].

The scalability and performance of a distributed DBMS depends on the number of transactions that access

data on multiple nodes. Whether or not a transaction needs to access multiple nodes depends on the design (i.e., physical layout) of the database. Such a design defines how an application's data and workload is partitioned or replicated across DBMS's nodes in a cluster, and how queries and transactions are routed to nodes. This in turn determines the number of transactions that access the data stored on each node and how skewed the load is across the cluster. But the problem of generating the optimal database design is known to be NP-Complete [164, 176], and thus it is not practical to examine every possible design to find one that works well [230].

It is also not possible for some applications to eliminate distributed transactions entirely through careful partitioning. This is either because of the application's workload properties (e.g., the database is not perfectly partitionable) [113] or for regulatory reasons (e.g., customers' financial data must be partitioned by their country of residence). Furthermore, decomposing distributed transactions into multiple single-partition transactions [173] are not extensible or may not provide the correct level of isolation [45].

Even if the application has a database design that minimizes the number of distributed transactions in its workload, the DBMS must still know at runtime what each transaction request will do when it executes. This information enables the system to apply certain optimizations to execute transactions more efficiently [180]. But it is not practical to require administrators to explicitly inform the DBMS how individual transaction requests are going to behave. Transaction invocations of the same stored procedure may not always access data at a single partition depending on the state of the database or the values of their input parameters. This is especially true for complex enterprise applications where a change in the database's configuration can affect transactions' execution behavior. This non-determinism makes it difficult for the DBMS to select the proper optimizations at runtime [72, 193]. Thus, the DBMS needs to infer the runtime behavior of transactions before they start running to overcome these impediments. It needs to be able to do this quickly and in a decentralized manner (e.g., the DBMS cannot spend 100 ms figuring out this information for a transaction that will only run for 5 ms).

1.4 Summary of Goals & Contributions

The above discussion illustrates the challenges of high-performance transaction processing. That is, just because a database is deployed on a distributed or main memory DBMS, does not mean that it will automatically perform better than a traditional DBMS on a single node. Thus, the overall goal of this dissertation is to improve the state-of-the-art in high-performance transaction processing by solving the issues identified above. This effort brings together a rich history of previous distributed DBMSs, main memory DBMSs, and transaction processing systems. Our work also incorporates techniques from the optimization and machine learning research communities in order to overcome the inherent scaling problems in DBMSs for OLTP applications. In essence, we hope that this document serves as a guide for the future endeavors of others in building a distributed, main memory OLTP DBMS.

The contributions in this dissertation are as follows:

Distributed, Main Memory DBMS Architecture for OLTP Applications: We provide a detailed description of our implementation of a new DBMS that is designed for the efficient execution of transactions in modern OLTP workloads. This system is a distributed, row-storage relational OLTP DBMS that runs

on a cluster of shared-nothing, main memory-only nodes [133, 211]. Such a clean slate architecture for OLTP applications has been referred to as a *NewSQL* system [23].

Automatic Database Design: We present an automatic tool for generating a near-optimal database design that partitions an OLTP database in such a way that minimizes communication overhead of transactions and manages skew [181].

Predictive Transaction Modeling: We present a method to automatically predict the runtime behavior of individual transactions using a low-impact machine learning framework integrated in the DBMS [180]. The DBMS then uses this information to enable various optimizations.

Speculative Transaction & Query Execution: We present a technique to improve the DBMS's performance through aggressive speculative execution of transactions and queries whenever a distributed transaction stalls waiting for messages to arrive over the network. This speculative execution allows the DBMS to perform useful work when it would otherwise be idle.

All of our contributions in this dissertation are in the context of the *H-Store* [3] DBMS that we developed. Although we use the H-Store system as our experimental test-bed in our analysis, the concepts discussed here are applicable to similar DBMSs.

The outline of this dissertation is as follows. We begin in Chapter 2 with a more thorough discussion of traditional DBMS architectures and demonstrate how they are insufficient for OLTP workloads. This analysis serves as a motivation for our new DBMS architecture that is optimized for modern applications. We then discuss in Chapter 3 the internals of the H-Store system that we built to explore and validate this proposed architecture. We then present three different optimizations that allow a distributed, main memory DBMS system to scale its performance. In Chapter 4, we discuss our automatic database design algorithm. Then in Chapter 5, we discuss our approach for integrating a machine learning framework in the DBMS to automatically derive execution properties for transactions. Such information is used to selectively enable runtime optimizations for individual transactions. We then build on this idea in Chapter 6 and show how this framework allows the DBMS to speculatively execute other transactions and queries whenever the DBMS waits for network communication. We conclude with a discussion of the related work in Chapter 7 and future work in Chapter 8.

Chapter 2

High-Performance Transaction Processing

The architecture of a traditional DBMS is inimical to the scaling demands of modern, high-performance transaction processing workloads [112, 127]. Many of the problems are due to the assumption in a traditional DBMS that the database is stored primarily on disk. But one cannot just store a database in main memory using a traditional system (e.g., with a RAM disk) and expect that an application will immediately perform better. As we discuss in this chapter, these DBMSs are comprised of tightly-coupled, disk-oriented components that are not designed for main memory storage and cannot be avoided without a complete rewrite of the system.

To better appreciate the problems of these components, we begin with a simple example that measures how well traditional DBMSs scale when they are provided with more CPU cores. We then examine another study that measures how much time the CPU spends in a traditional architecture's components when executing transactions. The results from our first example and the findings from the overhead study motivate the creation of a new DBMS architecture for high-performance transaction processing systems.

2.1 Motivating Example

The Voter benchmark is based on the transaction processing system used for two talent shows on Japanese and Canadian television (cf. Section A.7). The format of this type of show is the same in both countries. Contestants appear on stage and perform some act, and then viewers either call in or go on-line to vote for their favorite contestant. For each vote, the DBMS executes a transaction that first checks to see how many times the user has voted in the past, and then inserts a new vote entry for the contestant and updates a vote counter. While the show is on the air, viewers are shown a tally of the number of votes that each contestant has received. Since the results are needed in "real-time", the system must process votes as they arrive. This means that it cannot write votes to a log file and then run a batch program to compute the vote count at the end of the day. This transaction processing use case is known as a "leader board" and is also common in on-line gaming applications [122].

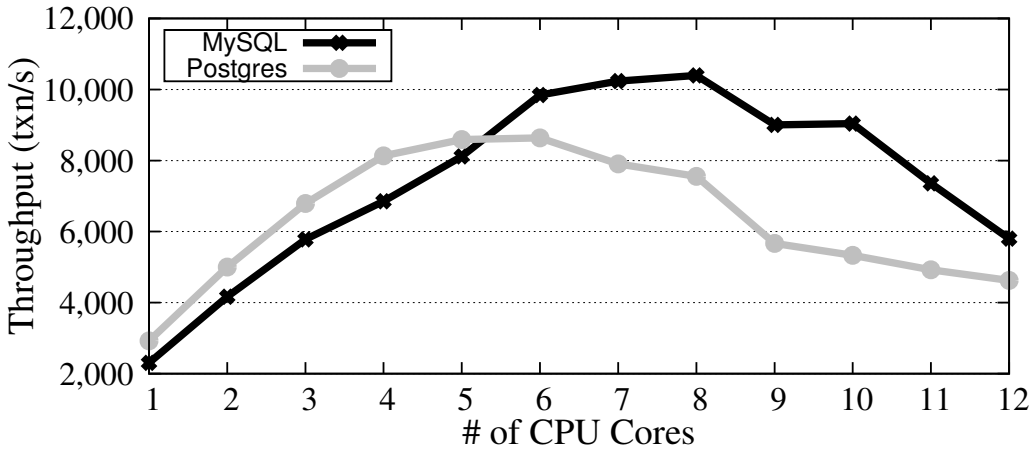


Figure 2.1: The throughput measurements for the Voter benchmark on MySQL and Postgres.

The nature of this type of television program also means that the arrival rate of transactions is inherently "bursty." Although the total number of transactions executed during a one-hour episode is not large (i.e., only several million requests per hour), the arrival rate of requests is heavily skewed at multiple intervals during that hour. This is because viewers are more likely to vote immediately after a contestant performs and each viewer is allowed to vote multiple times (up to a limit). Thus, the purpose of this benchmark is to measure the DBMS's peak performance by saturating it with many short-lived transactions that all update the same small number of records.

To evaluate how well traditional DBMSs perform on this workload, we ran an experiment using the Voter benchmark on two popular, open-source systems: MySQL (v5.6) [6] and Postgres (v9.1) [208]. We tuned each DBMS for OLTP workloads according to best-practice guidelines. The systems were deployed on a node with an Intel Xeon E7-4830 CPU running 64-bit Ubuntu Linux 12.04. The data for each respective DBMS was stored on a single 7200 RPM disk drive. According to *hdparm*, this disk delivers 7.2 GB/sec for cached reads and about 297 MB/sec for buffered reads. All transactions were executed with a serializable isolation level.

For each trial, we increase the number CPU cores that we allocate to the DBMS to process transactions. This allows us to measure how well these systems scale when they are provided more resources. Each DBMS is provided with enough DRAM to keep the entire database in main memory. This particular benchmark only touches a small number of tuples, so giving the systems more DRAM beyond this amount will not affect their runtime performance. For each CPU configuration on a DBMS, we ran the benchmark three times and report the average results. Transaction requests are submitted from up to 100 clients running on a separate node in the same cluster. Each client submits transactions to any DBMS node in a closed loop (i.e., it blocks after it submits a request until the result is returned). In each trial, the DBMS "warms-up" for 60 seconds and then the performance metrics are collected for five minutes. The throughput results are the number of transactions completed divided by the total time (excluding the warm-up period).

The results in Fig. 2.1 show that both MySQL and Postgres are unable to scale past 12,000 transactions a second. In the case of Postgres, the performance degrades as the system is allotted more CPU cores; at 12

cores, its throughput is 46.4% lower than its peak of ∼8,600 transactions per second at six cores. Likewise, MySQL's throughput drops by 44.2% from its peak of ∼10395.3 transactions per second at eight cores. Both of these performance degradations are due to increased lock contention of the DBMSs' shared resources at the larger core counts. As such, these peak throughputs for both DBMS's is insufficient.

2.2 Traditional DBMS Architectures

To understand why these DBMSs are unable to scale for what is a seemingly simple workload like the Voter benchmark, we need to discuss the key components of traditional, disk-oriented DBMS architectures. As discussed in Section 1.2, the design of traditional DBMSs is based on the computing hardware that was prevalent in the 1970s when the amount of memory available in typical a node was small. For these systems, disks were the primary storage location of databases because they were able to store more data than could fit in memory and were less expensive.

Many of the architectural components used in the early DBMSs are based on this hardware assumption. For example, traditional DBMSs assume that the node does not have enough memory to store the entire database, thus they use a *buffer pool manager* to store and manage data in heavily-encoded blocks stored on disks [24, 52, 209]. They also assume that a transaction could run for a long time (e.g., the transaction may have to wait for additional input from a human operator). Therefore, these systems employ a *concurrency control scheme* to allow multiple transactions to run simultaneously to keep the system busy in the event of a stall. These systems also rely on a complex *recovery mechanism* that allows them to quickly restore the database in the event of a crash [161]. Recovery speed was important because it was assumed that the database was managed by only one node. As a result, supporting high throughputs and low latencies for a large number of concurrent users is difficult [198, 201].

But because large-memory computers are now affordable, using a DBMS that is oriented towards storing data on slow disks does not make sense. To quantify the overhead of the obsolete components in traditional DBMS architectures, we now discuss the study by Harizopoulos et al. that measured the CPU instructions in a DBMS when it executes transactions [112]. For this evaluation, they instrumented the open-source Shore DBMS [47] to allow them to measure the overhead of the system's (1) buffer pool, (2) concurrency control scheme, and (3) recovery mechanism. They used the `NewOrder` transaction from the TPC-C benchmark [216] with a database that is stored entirely in main memory.[1] Since they only counted CPU instructions, this experiment only measures the processing time of the system and not the time needed to retrieve data.

As the results in Fig. 2.2 show, these three traditional architecture components account for ∼88% of the total CPU time in the DBMS [112]. That means that only ∼12% of the CPU's instructions is spent processing transactions (e.g., executing queries, invoking the transaction's program logic). The additional contention caused by multiple CPU cores cause the DBMS's threads to spend more time trying to acquire locks for shared data structures (e.g., B-trees) than actually executing transactions. This explains why the systems used in the experiments in Section 2.1 perform worse as they use more CPU cores. These findings are not isolated to Shore, but rather are similar to findings in other studies on running OLTP workloads on traditional DBMSs [127, 128, 220].

[1] The `NewOrder` transaction performs the same set of operations as the transactions in the Voter benchmark (i.e., read some data, insert new tuples, and update existing tuples).

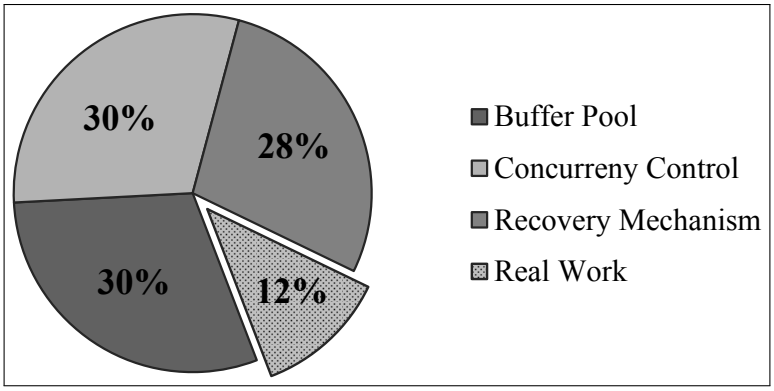


Figure 2.2: The percentage CPU instructions in the Shore DBMS when executing the `NewOrder` transaction from the TPC-C benchmark [112].

We now discuss the results from the study in [112] for the three architecture components.

2.2.1 Buffer Pool Management

In a traditional DBMS architecture, the system maintains a buffer pool of disk blocks in main memory. This provides a cache for recently retrieved records to speed up access times. When the application invokes a query that attempts to read a disk block, the system first checks to see whether that block already exists in this buffer pool. If not, then the DBMS retrieves the block from disk, translates it into a main memory format (called "swizzling" [62]), and then copies it into the buffer pool. If the buffer pool is full (i.e., there is no more space to store the new block), then the DBMS must evict a block from the pool to make room for the needed one. In order to decide what block to evict, the DBMS maintains an eviction order policy (e.g., least recently used) to increase the likelihood that it evicts blocks that will not be needed by future queries [92].

The results in Fig. 2.2 show that maintaining all of this information for the buffer pool accounts for $\sim$30% of the CPU instructions in Shore. This result is significant because these instructions represent completely wasted work. That is, since the database was already in memory, the DBMS never needed to retrieve a block from disk and it never had to evict a block from the buffer pool in order to make room for a new block. Any DBMS based on the traditional architecture will always incur this unnecessary overhead to check whether a block is in the buffer pool, regardless if the entire database is in main memory.

2.2.2 Concurrency Control Scheme

Whenever a transaction needs a block that is not in the buffer pool, the DBMS will stall that transaction until that block is retrieved. The amount of time that it takes to fetch a block from stable storage can be large. For example, the latencies for a read or write operation in a hard-disk drive storage device is approximately $\sim$3 ms/op [186, 190]. Although newer solid-state storage devices are faster (i.e., $\sim$0.025 ms/read, $\sim$0.2 ms/write), they are still much slower than DRAM (i.e., $\sim$0.000055 ms/op). Transactions could also stall if the DBMS has to wait for the application to send the next query to execute. This conversational-style transaction interface is commonly used in Web-based, application frameworks.

To mask high disk latency and network stalls, traditional DBMSs execute multiple transactions simultaneously. Now when one transaction stalls because the block that it needs is not in the buffer pool, the DBMS services other transactions while that block is retrieved. The DBMS's concurrency control scheme ensures that transactions execute with serializable consistency [35, 93], meaning that each transaction effectively executes as if no other transaction is running at the same time. One of the first concurrency control scheme for DBMSs was the multi-granularity two-phase locking protocol from the System R project [98]. This type of locking is the basis of all pessimistic concurrency control schemes that block transactions if there is a possibility that a non-serializable schedule could occur [115]. That is, if a transaction needs to acquire a lock for a particular tuple that is currently held by another transaction, then the first transaction has to wait until the other transaction releases that lock. To ensure that transactions are not blocked waiting for locks indefinitely, the DBMS can either run a deadlock detection algorithm or restart transactions if they stall for longer than an administrator-defined threshold. Other optimistic approaches [138], such as multi-version concurrency control [31, 53], continue to process transactions even when a potential conflict occurs and then checks whether serializability was compromised when the transactions complete. Regardless of which approach a DBMS uses, a concurrency control scheme adds overhead to transaction processing.

Concurrent transactions also complicate the management of the DBMS's buffer pool. When a transaction stalls because it needs a block on disk, the DBMS has to pin all the other blocks that are already in memory that the transaction needs to ensure that they are not evicted while the system waits for the new block. This means that the DBMS must protect the internal data structure it uses to track pinned blocks with more locks to ensure that it is not corrupted by concurrent modifications.

The runtime measurements from Shore in Fig. 2.2 indicate that the overhead due to its concurrency control scheme is non-trivial. The DBMS's lock-based scheme accounts for $\sim$30% of the total CPU instructions during transaction execution [112]. This expense from coordinating multiple transactions is again unnecessary because the transactions never stalled waiting for a block on disk or waiting for the next query (i.e., all of the program logic for the transactions were embedded in the system). This result strongly suggests that the heavy-weight concurrency control schemes used in traditional DBMSs are unnecessary. Some disk-oriented DBMSs use small lock granularities to reduce contention, but the benefit of using finer-grained locks is small if the contention is already low because all of the data is in memory and transactions are short-lived. Instead, a main memory DBMS is better off using larger lock granularities [148, 151, 192]. That is, if the cost of acquiring a lock in memory is the same as just accessing a tuple in memory, the transaction is better off just accessing the tuple. This removes almost all of the cost of concurrency control and reduces the number of CPU cache flushes that result from transaction context switches [88].

2.2.3 Recovery Mechanism

Every enterprise-quality DBMS ensures that all of a transaction's modifications to the database are durable and persistent [115]. But they do not write each individual change made to the database to disk separately, since this would likely involve multiple random writes to the disk per transaction. This approach is also problematic if the DBMS crashes in the middle of the transaction because when the system comes back online the database will be in an inconsistent state (i.e., the transaction's updates were not atomic). Instead, all DBMSs employ a log-based recovery mechanism that stores the changes made by transactions separately so

it can restore the database to the proper state after a crash.

In a traditional architecture, like the one used in Shore, the DBMS updates blocks in the buffer pool and then maintains a separate *write-ahead log* for recovery that is written to disk [161]. These changes are written out in batches to amortize the cost of syncing the disk [114]. After a crash, this log is first processed backwards to undo the effects of uncommitted transactions, and then again in the forward direction to redo the effects of committed transactions [115].

Since the DBMS does not write changes to tuples immediately to disk, it will keep those blocks in the buffer pool but mark them as "dirty." Now when the DBMS needs to evict a block from its buffer pool to make room, it has to check this flag for each block selected for eviction. If the selected block does not have the dirty flag enabled, then the DBMS can simply discard it. Otherwise, the system will need to write that block out to disk. This means that the DBMS essentially maintains the database state in two separate locations on disk: in the primary storage of blocks and in its log files. Other approaches, such as log-structured merge trees [172], seek to resolve this dichotomy by using just the log to represent the state of the database.

The need to store a log of the transactions' modifications on stable storage can undermine the performance of a high-performance transaction processing system, even if the database is in main memory [88, 128]. Similarly, it can also affect the system's response time because each transaction must wait for at least one write to stable storage before they can commit. Although technologies like battery-backed memory controllers and uninterruptable power supplies are available, there will still need to be a copy of the database on disk. But the cost the preparing the undo and redo log entries in traditional DBMS recovery mechanisms is non-trivial [161]. In the case of the results shown in Fig. 2.2, this accounts for ∼28% of the CPU instructions in Shore. Much of this is due to the preparation of writing the DBMS's data structures into log entries. This data must be protected so that the log entries are transactionally consistent. Hence, a main memory-oriented DBMS will need a recovery scheme that is optimized for main memory-resident data and can avoid this preparation overhead [128, 150, 155].

2.3 Strategies for Scaling DBMSs

Given the significant overhead of disk-based DBMS architectures, there are several different approaches that one can use to improve the throughput and responsiveness of a DBMS for OLTP workloads. The first strategy that most organizations usually take is to migrate the database to a node with faster, more powerful hardware. Although moving to a faster node may provide an immediate performance boost, it is not a long-term solution. If the number of requests continues to increase after an upgrade, then the database will need to be moved again to even faster hardware. But every upgrade requires a difficult migration from the old node to the new one that may require significant downtime [87]. Previous studies have also shown that the performance of a single-node, traditional DBMS does not always improve with each upgrade due to increased lock contention [112, 128, 187].

Another common approach is to replicate the database in its entirety on multiple nodes [97, 117]. The most widely used replication scheme is to designate one DBMS node as the master copy that receives all update requests from the application. This master then streams modifications out to slave nodes (i.e., replicas). If the master node crashes, then the system will use a consensus protocol, such as Paxos [142], to elect a new

master. Using this type of replication scheme will improve the latency of read-only requests, since they can now be serviced by the replica nodes. But it will slow down the performance of write requests because the master node has to wait for the slaves to acknowledge that they received the changes before it can send a message back to the application that the modification was successful [97].

Organizations can also deploy a main memory distributed cache to minimize the load on the DBMS [85]. Under this two-tier architecture, the application first looks in the cache for the tuple of interest. If this tuple is not in the cache, then the application executes a query in the DBMS to fetch the desired data. Once the application receives this data from the DBMS, it updates the cache for fast access in the future. Whenever a tuple is modified in the database, the application must invalidate its cache entry so that the next time that the tuple is accessed the application will retrieve the current version from the DBMS. This is similar to replicating the database to speed up read-only requests except that it requires developers to embed logic in their application to keep the two systems independently synchronized. For example, when an object is modified, the update is sent to the back-end DBMS. But now the states of the object in the DBMS and in the cache are different. If the application requires up-to-date values, the application must also update the object in the cache.

The next scaling tactic is to use custom middleware that resides in between the application and multiple, single-node DBMSs [198, 201]. Each DBMS node contains a subset of the database, called a *shard*, and the middleware routes each query from the application to the shard containing the data that it needs. This allows the organization to combine multiple, single-node DBMSs together to appear as a single DBMS instance to the application. This also makes it easier to expand capacity because a new node can be added into the system without potentially needing to copy data from other nodes. The downside of this approach, however, is that certain functionalities, such as joins, must be implemented in application code and more advanced features, such as transactions, are notoriously difficult to implement correctly. This approach is also prohibitively expensive for most organizations, as it requires their developers to build infrastructure rather than working on the applications directly related to their business.[2]

Finally, the most momentous change in scaling large-scale applications is to not use a relational DBMS at all and instead use a *NoSQL* system [49]. These systems are noted for providing better scalability and availability that traditional, monolithic DBMSs. NoSQL systems achieve this better performance, however, by forgoing much of the ACID guarantees of traditional DBMSs. For example, many NoSQL systems only provide eventual consistency guarantees or limit atomic operations to single records [54, 69]. This approach is desirable if the consistency requirements of the data are "soft" (e.g., status updates on a social networking site that do not need to be immediately propagated throughout the application) [222]. These trade-offs made in NoSQL DBMSs are appropriate for some situations, but they are unsuitable for many applications that need multi-operation transactions and serializability [137]. Such systems are too unwieldy for these applications because they again require developers to re-implement the features that are normally provided by a DBMS, such as atomicity and consistency [30, 95].

[2]We note that there are now several companies that offer middleware DBMS products, but these were not available in the early 2000s when companies like eBay [198] and Facebook [201] famously built their own large-scale sharding middleware.

2.4 A Scalable DBMS Architecture for OLTP Applications

Our analysis in Section 2.2 shows that traditional DBMSs are insufficient for modern transaction processing systems [88]. And although the scaling solutions described above are inadequate by themselves, there are several aspects of them that are desirable in a DBMS architecture if they are combined together. Thus, what is needed is a new class of distributed DBMSs [70, 95] that provide the same scale-out capabilities of the middleware and NoSQL approaches, the high-availability of a replicated deployment, the high-performance of an in-memory cache, all while still maintaining the ACID guarantees of single-node, relational DBMSs [23].

To achieve this, we propose a new DBMS architecture that is comprised of five design principles:

Main Memory Storage: The DBMS uses main memory as the primary storage location for the database. This is different than just using a disk-oriented DBMS with a large buffer pool. For example, the indexes in a disk-oriented DBMS will still be designed for disk access [88], whereas a main memory-oriented DBMS can exploit cache locality [158].

Distributed Deployments: In order to support databases that are larger than the amount of memory available on a single node, the DBMS will split databases across shared-nothing [206] compute nodes into disjoint segments called *partitions* [49, 73]. This differs from traditional DBMSs, where multi-node support was added as an afterthought [160]. Partitions will be replicated across multiple nodes to provide the transaction processing system with high-availability and fault-tolerance.

Stored Procedure Execution: Rather than sending SQL commands at runtime, the application registers a set of SQL-based procedures with the DBMS and only invokes transactions through these procedures. Encapsulating all transaction logic in a single stored procedure prevents application stalls mid-transaction and also avoids the overhead of transaction parsing at run-time. Although this scheme requires all transactions to be known in advance, this assumption is reasonable for OLTP applications [155, 211].

Serial Execution: Instead of allowing transactions to execute concurrently, the DBMS executes transactions serially at each partition [148, 151, 192, 224]. This means that it does not need to employ a heavy-weight concurrency control scheme to manage fine-grained locks. When a transaction executes, it never has to stall because all of the memory that it needs is already in main memory and it will never block waiting to acquire a lock held by another transaction.

Compact Logging: To avoid the overhead of a heavy-weight recovery mechanism [161], the DBMS uses a lightweight logical logging scheme that only needs to record what transactions were rather than the individual physical changes that it made to the database.

A DBMS based on this new architecture will avoid all of the overhead described in Section 2.1 from (1) buffer pool management, since everything is in memory, (2) heavy-weight concurrency control schemes, since transactions execute serially, and (3) creating the physical undo/redo log entries for transactions, since the DBMS only records logical log entries. We note that we do not intend this to be a general purpose DBMS architecture. Rather than try to have the system perform somewhat well for a variety of workloads, we restrict our focus to a specific application domain that are amenable to partitioning. We believe that

part of the reason why earlier distributed DBMSs were unsuccessful were that they were targeting general transactional workloads [113, 212, 225].

We next describe the system that we built, called H-Store [3], which embodies these design principles. We then describe in Chapters 4 to 6 the different optimizations that we developed to ensure that the system can scale its performance for larger cluster sizes.

Chapter 3

The H-Store OLTP Database Management System

In the previous chapter, we demonstrated how the performance of OLTP workloads does not scale in traditional DBMSs even if the database fits entirely in main memory. This is because of the overhead of the legacy architectural components in a disk-oriented system [20, 112]. We then proposed a new architecture for a distributed, main memory DBMS that targets modern hardware and OLTP workloads.

We have developed a new DBMS from the ground up based on this proposal, called *H-Store* [3], that is designed for the efficient execution of transactions in OLTP applications [133, 211]. The H-Store project is a collaboration between Brown University, MIT, and Yale University.

Over the years, there have been several incarnations of the H-Store system. The initial proof-of-concept was a single-node engine developed at MIT in 2007 that only could execute a simplified version of the TPC-C benchmark [211]. The full-featured, general purpose version of H-Store was developed in 2008 by Brown, MIT, Yale, and Vertica Systems [133]. In 2009, this version of H-Store was forked and commercialized as VoltDB [10]. In 2010, some of the changes from VoltDB were merged back into the original H-Store source code, but we rewrote the runtime transaction management and coordination subsystems. Since then, H-Store's development has occurred primarily at Brown by the author of this dissertation in order to create a test-bed for the experimental results presented here. As of 2013, many of the core components of the original H-Store system still exist in VoltDB (e.g., the underlying execution engine).

We now describe the internals of H-Store's architecture. We begin with an overview of the terminology and concepts that are used throughout this dissertation. Then in the subsequent sections, we describe H-Store's main components and discuss how the system operates at runtime.

3.1 Overview

We define a single H-Store instance as a *cluster* of one or more shared-nothing [206] nodes deployed within the same administrative domain. A *node* is a single physical computer system that hosts an H-Store process

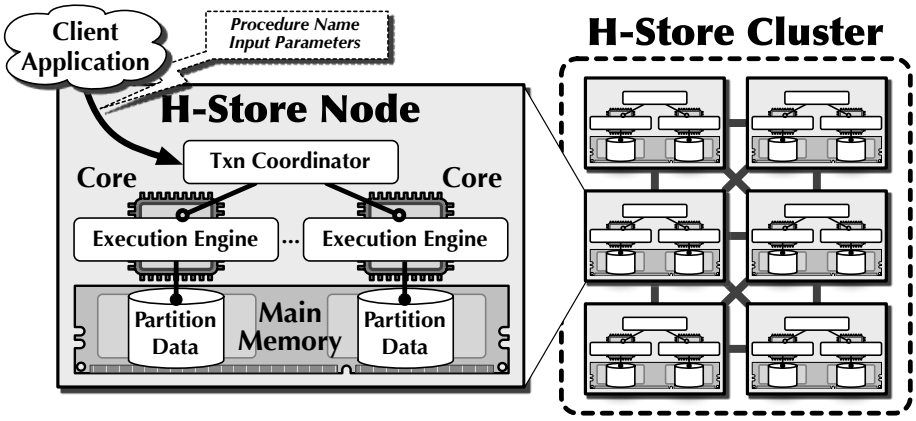


Figure 3.1: An overview of the H-Store distributed OLTP DBMS. Each H-Store node consists of a transaction coordinator that manages single-threaded execution engines, each with exclusive access to a data partition stored in memory. All tuples in H-Store are stored in main memory replicated on multiple nodes and all transactions are executed as pre-defined stored procedures.

that contains one transaction coordinator that manages one or more partitions. We assume that the typical H-Store node contains multiple CPU cores. H-Store is designed to operate on commodity hardware. This makes it easier and more affordable to scale the database out across multiple nodes to increase both the DBMS's CPU and memory capacity.

The *transaction coordinator* is how the DBMS at one node communicates with other nodes in the cluster. H-Store's transaction and cluster management subsystem is completely decentralized. Collectively, the coordinators are responsible for ensuring that the DBMS executes transactions in the correct order, execute queries and process commit protocol messages for multi-partition transactions, and to perform system administrative functions.

A *partition* is a disjoint subset of the entire database [109, 230]. Each partition is assigned a single-threaded *execution engine* [224] that is responsible for executing transactions and using the coordinator to communicate with other execution engines. Each execution engine has exclusive access to the data at its partition. That means that if a transaction needs to access data at a partition, then it has to get added to the queue for that partition's engine and wait to be executed by it.

All tuples in H-Store are stored in main memory using a row-oriented format. The DBMS replicates partitions to ensure both data durability and availability in the event of a node failure. Data replication in H-Store occurs in two ways: (1) replicating partitions on multiple nodes and (2) replicating an entire table at all partitions (i.e., each partition has a complete copy of the table). For the former, we adopt the k-*safety* concept, where k is defined by the administrator as the number of node failures a database can tolerate before it is deemed unavailable [133]. We discuss partition replication in Section 3.8 and table replication in Section 3.4.2.

We assume that the majority of the application's transactions are executed in H-Store as pre-defined *stored procedures*. Although H-Store supports ad hoc transactions that do not use stored procedures, such transactions are not the common case and thus the system is not optimized for them [48]. Each stored

procedure is comprised of (1) parameterized queries and (2) control code that contains application logic intermixed with invocations of those queries (cf. Fig. 3.3). We use the term *transaction* in the context of H-Store to refer to an invocation of a stored procedure. Client applications initiate transactions by sending the procedure name and input parameters to any node in the cluster. The location where the transaction's control code executes is known as its *base partition* [180]. The transaction's base partition will have most (if not all) of the data the transaction needs. Any other partition that is involved in the transaction that is not its base partition is referred to as a *remote partition*.

3.2 System Architecture

H-Store's architecture is divided into two parts: (1) the front-end transaction management component and (2) the back-end query executor and storage manager. The front-end consists of all the networking libraries for communicating with the application's clients, the transaction coordinator, and the stored procedures. This part of the system is written in Java. The back-end C++ execution engine for each partition contains the storage manager, indexes, and query plan executors. This part of the system is responsible for managing all of the data in the system.

3.2.1 Transaction Coordinator

When H-Store starts, each node is provided with a list the host-names and port numbers of the other nodes in the cluster. Each node's transaction coordinator then establishes a TCP/IP connection with these nodes. We use Google's Protocol Buffer serialization library to construct the coordinator's messages and invoke remote operations over the network. The coordinators also send periodic "heartbeats" to each other to let them know that the DBMS is operating correctly at their corresponding node.

Instead of using a heavy-weight concurrency control scheme where multiple transactions execute simultaneously at a partition [29], H-Store executes transactions one-at-a-time at each partition. That is, when a transaction executes in H-Store, it has exclusive access to the data and indexes at the partitions that it needs. Transactions never stall waiting to acquire a latch held by another transaction because no other transaction will be running at the same time at either its base partition or its remote partitions.

H-Store uses timestamp-based scheduling for transactions [29]. When a transaction request arrives at a node, the coordinator assigns the request a unique identifier based on its arrival timestamp. This id is a composite key comprised of the current wall time at the node (in milliseconds), a counter of the number of transactions that have arrived since the last tick of the wall time clock (in case multiple transactions enter the system at the exact same time), and the transaction's base partition id [217].

Each partition is protected by a single lock managed by its coordinator that is granted to transactions one-at-a-time based on the order of their transaction ids [15, 29, 64, 224]. A transaction acquires a partition's lock if (1) the transaction has the lowest id that is not greater than the one for last transaction that was granted the lock and (2) it has been at least 5 ms since the transaction first entered the system [211]. This wait time ensures that distributed transactions that send their lock acquisition messages over the network to remote partitions are not starved. We assume that the standard clock-skew algorithms are used to keep the various CPU clocks synchronized at each node.

Serializing transactions at each partition in this manner has several advantages for OLTP workloads. In these applications, most transactions only access a single entity in the database at a time (e.g., a transaction that operates on a single customer). That means that if the DBMS will perform significantly faster than a traditional DBMS if the database is partitioned in such a way that most transactions only to access a single partition. Smallbase was an early proponent of this approach [118], and more recent examples include K [224] and Granola [64]. The downside of this approach, however, is that it means transactions that need to access data at two or more partitions are significantly slower. If a transaction attempts to access data at a partition that it does not have the lock for, then the DBMS aborts that transaction (releasing all of the locks that it holds), reverts any changes, and then restarts it once the transaction re-acquires all of the locks that it needs again. Employing such an approach removes the need for distributed deadlock detection, resulting in better throughput for short-lived transactions in OLTP applications [112].

The coordinator queues the request at that all of the nodes that contain the partitions that the transaction will access. When the transaction acquires a partition's lock, the coordinator prepares an acknowledgement message to send back to the transaction's base partition. Once the transaction acquires all the locks that it needs from a node's partitions, the coordinator sends this acknowledgement. Once a transaction receives all of the lock acknowledgements for the partitions that it needs, the coordinator for its base partition schedules the transaction to run immediately on its base partition's execution engine [30, 46].

3.2.2 Execution Engine

Every partition in H-Store is managed by a single-threaded execution engine that has exclusive access to the data at that partition. An execution engine is comprised of two parts, one written in Java and one written in C++. In the Java-level component, the execution engine's thread blocks on a queue waiting for messages to perform work on behalf of transactions. This work can either instruct the engine to invoke a procedure's control code to start a new transaction or to execute a query plan fragment on behalf of a transaction running at another partition. Note that for the latter, H-Store's transaction coordination framework ensures that no transaction is allowed to queue a query request at an execution engine unless the transaction holds the lock for that engine's partition.

The execution engine's C++ library is where H-Store stores databases (cf. Section 3.2.3) and processes queries. The Java layer uses the Java Native Interface (JNI) framework to invoke the methods in the C++ library and passes it the query plan identifiers that the transaction invoked. This library is not aware of other partitions or nodes in the cluster; it only operates on the input that it is provided.

3.2.3 Storage Layer

The diagram in Fig. 3.2 shows an overview of this storage layout for H-Store's tables and indexes. All of the execution engines at a single node operate in the same address space, but their underlying partitions do not share any data structures. Each partition maintains separate indexes for the database tables that only contain entries for the tuples associated with that particular partition. This means that an execution engine is unable to access data stored in another partition at the same node.

The in-memory storage area for tables is split into separate pools for fixed-sized blocks and variable-length blocks. The fixed-size block pool is the primary storage space for the tables' tuples. All tuples are a

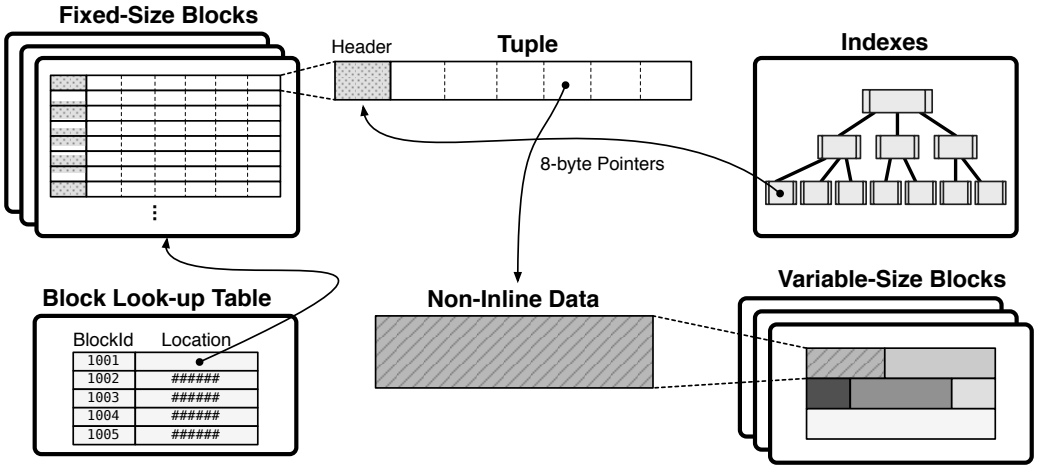


Figure 3.2: An overview of the in-memory storage layer for H-Store.

fixed size (per table) to ensure that they are byte-aligned. Any field in a table that is larger than 8-bytes is stored separately in a variable-length block. The 8-byte memory location of this block is stored in that field's location in the tuple [192]. All other fields that are less than 8-bytes are stored in-line. Each tuple is prefixed with a 1-byte header that contains meta-data on whether a tuple has been modified or has been deleted by the current transaction. This information is used for H-Store's snapshot mechanism (cf. Section 3.7.2).

The DBMS maintains a look-up table of the id numbers of blocks to their corresponding memory location. This look-up table allows the execution engine to reference individual tuples using a 4-byte offset in the table's storage area rather than an 8-byte pointer. That is, from a 4-byte offset the storage layer can compute the id of the block with the tuple and the tuple's location within that block.

The DBMS stores the tables' tuples unsorted within the storage blocks. For each table, the DBMS maintains a list of the 4-byte offsets of unoccupied (i.e., free) tuples. When a transaction deletes a tuple, the offset of the deleted tuple is added to this pool. When a transaction inserts a tuple into a table, the DBMS first checks that table's pool to see if there is an available tuple. If the pool is empty, then the DBMS allocates a new fixed-size block to store the tuple being inserted. The additional tuple offsets that are not needed for this insert operation are added to the table's free tuple pool. H-Store does not compact blocks if a large number of tuples are deleted from a table.[1] We note that such mass deletions are rare in OLTP workloads and do not occur in the workloads we used to evaluate H-Store in this dissertation (cf. Appendix A).

H-Store supports hash table and B-tree data structures for unique and non-unique indexes. The values of the entries in the indexes are offsets for tuples. We use the C++ standard template library implementations for these data structures; more optimized, cache-conscious alternatives are available [158]. For the B-tree indexes, the copies of key fields for each tuple are stored in the data structure.

Before a new application can be deployed on H-Store, the administrator has to provide the DBMS's *Project Compiler* with (1) the database's schema, (2) the application's stored procedures, and (3) the database's

[1]The commercial version of H-Store's design (VoltDB) supports automatic block compaction and reorganization

design specification. The compiler will generate a catalog that contains the meta-data for the components in the application's database (e.g., tables, indexes, constraints) and the compiled query plans for each of the application's stored procedures. We now describe these stored procedures in Section 3.3 and design specification in Section 3.4. We then discuss how the Project Compiler generates the query plans in Section 3.5.

3.3 Stored Procedures

Many OLTP applications utilize stored procedures to reduce the number of round-trips per transaction between the client and the DBMS [207]. Support for stored procedures first emerged in commercial DBMSs in the late 1980s and has been part of the ISO SQL standard for years. All major DBMS vendors support stored procedures today.

In H-Store, each stored procedure is identified by a unique name and consists of user-written Java *control code* (i.e., application logic) that invokes pre-defined parameterized SQL commands. The application initiates transactions by sending a request to the DBMS that contains the procedure name and input parameters to the cluster. The input parameters to these stored procedures can be either scalar or array primitive values.

As shown in the example in Fig. 3.3, a stored procedure has a "run" method that contains the application logic for that procedure. There are no explicit `begin` or `commit` commands for transactions in H-Store. A transaction begins when the execution engine of its base partition invokes this method and then completes when this method returns (either through the `return` or `abort` commands). When this control code executes, it makes query invocation requests at runtime by passing the target query's handle along with the input parameters for that invocation to the H-Store runtime API (e.g., `queueSQL`). The values of these input parameters will be substituted for the query's parameter placeholders (denoted by the "**?**" in the SQL statements in Fig. 3.3). The DBMS queues each invocation and then immediately returns back to the control code. Multiple invocations of the same query are treated as separately even if they use the same input parameters.

After adding all of the invocation requests that it needs to the current batch, the control code then instructs the DBMS to dispatch the batch for execution (e.g., `executeBatch`). At which point, the control code is blocked until the DBMS finishes executing all of the queries in the current batch or aborts the transaction due to an error (e.g., if one of the queries violates a integrity constraint). This command returns an ordered list of the output results for each query invocation in the last batch executed. We will discuss in Section 3.6.2 how H-Store determines what partitions each query invocation in the batch will access and how it uses its transaction coordinators to dispatch them to those partitions.

H-Store includes special "system" stored procedures that are built into the DBMS. These procedures allow users to execute administrative functions in the system, such as bulk loading data into tables, modifying configuration parameters, and shutting down the cluster.

Although stored procedures in H-Store contain arbitrary user-written Java code, we require that all of their actions and side-effects are deterministic. That is, each stored procedure must be written such that if the DBMS executes a transaction again with the same input parameters and in the same order (relative to other transactions), then the state of the database after that transaction completes will be the same. This means that the procedure's control code is not allowed to execute operations that may give a different result if it is executed again. This requirement is necessary for H-Store's replication scheme (cf. Section 3.8) and recovery

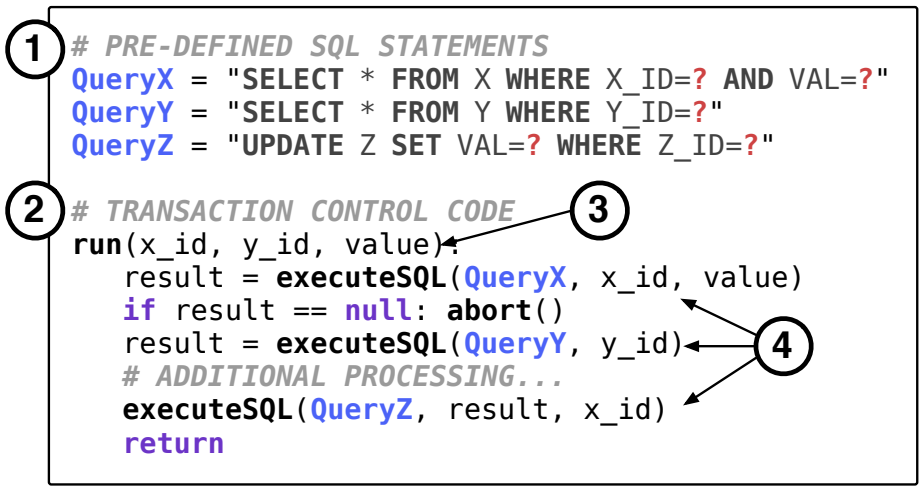


Figure 3.3: A stored procedure defines (1) a set of parameterized queries and (2) control code. For each new transaction request, the DBMS invokes the procedure's `run` method and passes in (3) the procedure input parameters sent by the client. The transaction invokes queries by passing their unique handle to the DBMS along with the values of its (4) query input parameters.

mechanism (cf. Section 3.7).

The types of non-deterministic operations that are forbidden in a stored procedure's control code include (1) using an RPC library inside of a procedure to communicate with an outside system, (2) retrieving the current time from the node's system clock, or (3) using a random number generator. As an example of why these are problematic, consider a procedure that contacts an outside third-party fraud detection system to determine whether a financial transfer is fraudulent during the transaction. The transaction will then choose whether to commit or abort based on the response from this system. One problem with doing this in an H-Store transaction is that this service may report a false positive if the same request is sent to it multiple times by different invocations of the same transaction running on different nodes (i.e., replicated deployments). Additionally, the service may be unavailable at a later date when the DBMS replays the transaction (i.e., crash recovery). In either case, the database will be inconsistent. Thus, in order for this application to work reliably in H-Store, the developer would need to move the fraud detection operation outside of the procedure.

3.4 Database Design Specification

An application's database design specification defines the physical configuration of the database, such as whether to divide a particular table into multiple partitions or to replicate it at every node. This determines the partitions that each transaction will access at runtime. A design determines whether the H-Store executes a transaction request from the application as a fast, single-partition transaction, or as a slow, distributed transaction. That is, if tables are divided amongst the nodes such that a transaction's base partition has all of the data that the transaction needs, then it is single-partitioned [66, 181].

We now describe the four components of a database design in H-Store. The administrator must provide this information to the system before it can be started. But determining the optimal configuration for an

arbitrary application is non-trivial, especially for a complex enterprise application with many dependencies. Thus, in Chapter 4 we discuss how to automatically compute a database design that minimizes the number of distributed transactions and the amount of skew.

3.4.1 Table Partitioning

A table can be horizontally divided into multiple, disjoint fragments whose boundaries are based on the values of one (or more) of the table's columns (i.e., the *partitioning attributes*) [230]. The DBMS assigns each tuple to a particular fragment based on the values of these attributes using either range partitioning or hash partitioning. Related fragments from multiple tables are combined together into a partition [90, 176]. Most tables in OLTP applications will be partitioned in this manner. In the example database in Fig. 3.4a, each record in the `CUSTOMER` table has one or more `ORDERS` records. Thus, if both tables are partitioned on their `WAREHOUSE` id (e.g., `CUSTOMER.W_ID` and `ORDERS.W_ID`), then all transactions that only access data within a single warehouse will execute as single-partitioned, regardless of the state of the database.

3.4.2 Table Replication

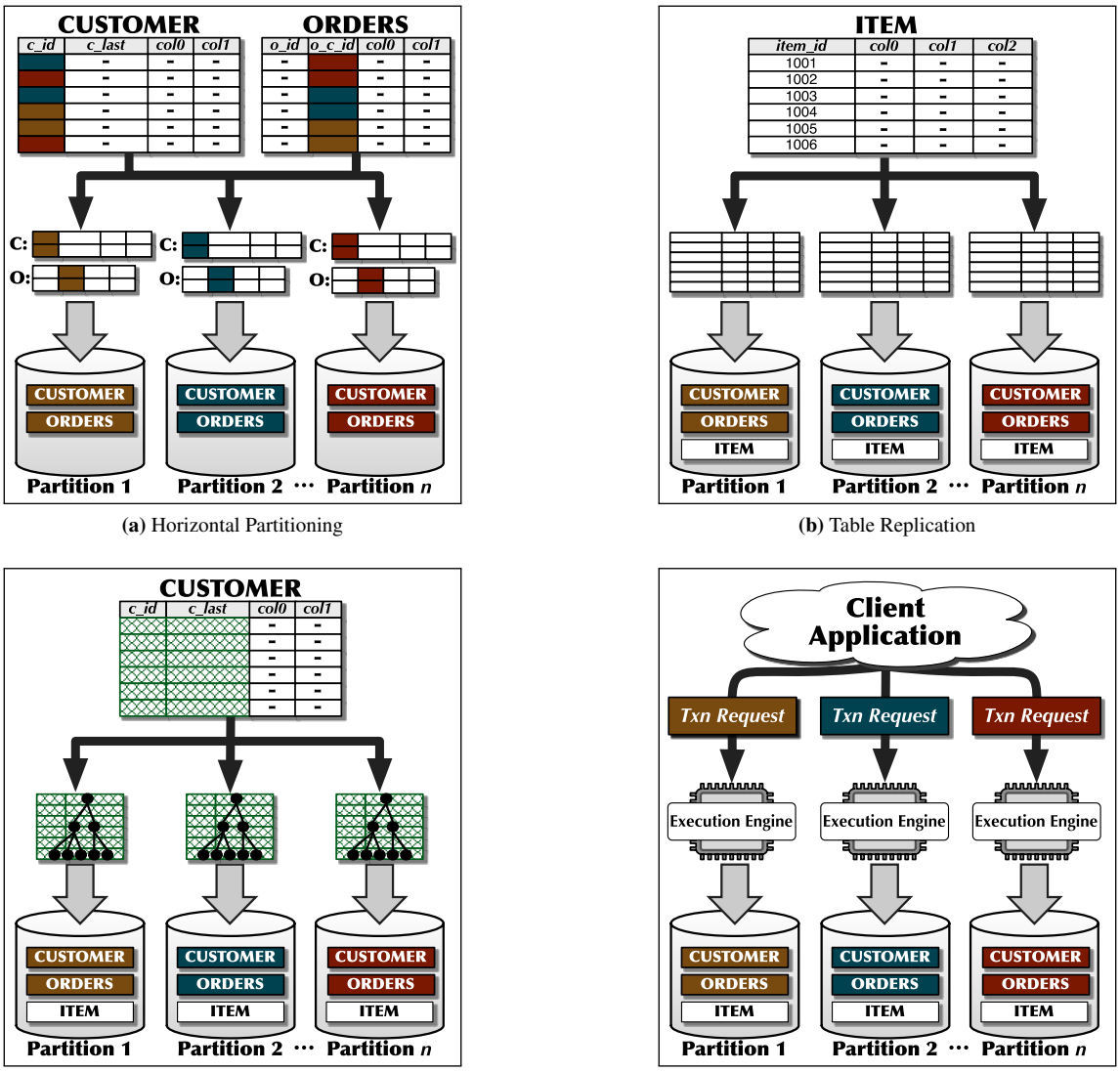
Instead of splitting a table into multiple partitions, the DBMS can replicate that table across all partitions. Table replication is useful for read-only or read-mostly tables that are accessed together with other tables but do not share foreign key ancestors. This is different than replicating entire partitions for durability and availability. For example, the read-only `ITEM` table in Fig. 3.4b does not have a foreign-key relationship with the `CUSTOMER` table. By replicating this table, transactions do not need to retrieve data from a remote partition in order to access it. But any transaction that modifies a replicated table has to be executed as a distributed transaction that locks all of the partitions in the cluster, since those changes must be broadcast to every partition in the cluster. In addition to avoiding additional distributed transactions, one must also consider the space needed to replicate a table at each partition.

3.4.3 Secondary Index Replication

When a query accesses a table using a column that is not that table's partitioning attribute, it is broadcast to all partitions. This is because the DBMS does not know what partition has the tuple(s) that the query needs. In some cases, however, these queries can become single-partitioned if the database includes a secondary index for a subset of a table's columns that is replicated across all partitions. Consider a transaction for the database shown in Fig. 3.4c that executes a query to retrieve the id of a `CUSTOMER` using their last name. If each partition contains a secondary index with the id and the last name columns, then the DBMS can automatically rewrite the stored procedures' query plans to take advantage of this data structure, thereby making more transactions single-partitioned. Just as with replicated tables, this technique only improves performance if the columns chosen in these indexes are not frequently updated.

3.4.4 Stored Procedure Routing

In addition to partitioning or replicating tables, a database design can also ensure that each transaction request is routed to the partition that has the data that it will need (i.e., its base partition) [184]. H-Store uses a procedure's *routing attribute(s)* defined in a design at runtime to redirect a new transaction request to a node



(a) Horizontal Partitioning

(b) Table Replication

(c) Secondary Index

(d) Stored Procedure Routing

Figure 3.4: A database design for H-Store consists of the following: (a) splits tables into horizontal partitions, (b) replicates tables on all partitions, (c) replicates secondary indexes on all partitions, and (d) routes transaction requests to the best base partition.

that will execute it [169]. The best routing attribute for each procedure enables the DBMS to identify which node has the most (if not all) of the data that each transaction needs, as this allows them to potentially execute with reduced concurrency control [181]. Fig. 3.4d shows how transactions are routed according to the value of the input parameter that corresponds to the partitioning attribute for the CUSTOMER table. If the transaction executes on one node but the data it needs is elsewhere, then it must execute with full concurrency control. This is difficult for many applications, because it requires mapping the procedures' input parameters to their queries' input parameters using either a workload-based approximation or static code analysis.

3.5 Query Plan Compilation & Optimization

Prior to deployment, H-Store's Project Compiler analyzes the application's stored procedures and generates the execution plans for each of their queries. This step is necessary because queries in the stored procedures do not have any information about how the database is partitioned or how many partitions there are in the cluster. This is desirable because it means that the developer does not need to consider the physical location of data when they write a procedure's control code and can therefore only interact with the database through declarative SQL queries. Compiling these plans before the system starts allows the DBMS to execute queries without needing to invoke the query planner for each transaction request at runtime [211].

A query plan is a directed tree that represents the steps that the DBMS will execute to process a query invocation [197]. The DBMS substitutes the parameters from the query using the input parameters that the procedure control code passed in when it invoked the query (cf. Fig. 3.5). At runtime, each operator in the tree is executed at one or more participating partitions for the transaction [73]. Each operator performs some computation and then produces an output table [70]. This output table is then passed as the input to the next operator in the tree as the DBMS continues processing the plan. An execution engine will not execute a vertex's operator until the vertex that generates its input finishes. The output table of the plan's root vertex is the final result of that query that is returned to the control code.

Since H-Store's query optimizer runs before the DBMS is brought on-line, it does not use statistics about the database's contents (e.g., table cardinalities) when generating query plans. This is not a problem for OLTP applications because the queries in these workloads are typically index look-ups on a single table. Thus, the optimizer can infer from a static analysis of a query's WHERE clause what the best index to use for it. But for those queries where the database's properties could change what query plan the DBMS chooses (e.g., the join order of two tables), the administrator can provide hints about the tables to the Project Compiler. In Section 8.7, we discuss alternative techniques for choosing the best query plan at runtime without needing to use the query planner each time.

When the Project Compiler processes an application's stored procedures, it examines their compiled Java class files and extracts their pre-defined SQL statements using Java's reflection API. The Project Compiler generates two types of query plans for each statement. One plan is optimized for execution at a single partition that the DBMS uses if a query invocation only needs to access data at one partition (even if it is a remote partition). The other plan type is optimized for execution on multiple partitions [81]. The multi-partition plan allows the DBMS to transfer data from one partition to another and is optimized to reduce the amount of network traffic [34, 175]. The Project Compiler generates both plan types regardless of whether a procedure will only ever execute as a single-partition transaction (e.g., if the procedure's queries only read from replicated tables).

Each query plan is assigned a unique identifier in H-Store's internal catalog. These identifiers are used in the messages sent between the coordinators at runtime; when a transaction invokes a query that will run on a remote partition, the coordinator sends a message that contains only this identifier and not the entire plan.

We now describe these two types of query plans and the optimizations that the Project Compiler applies to them. We will then discuss how H-Store determines at runtime which plan to execute for a query invocation at runtime in Section 3.6.2. A full description of H-Store's query plan operators and additional examples are

provided in Appendix B.

3.5.1 Single-Partition Query Plans

Consider the following SELECT query from the Payment transaction in the TPC-C benchmark. For this example, assume that the database uses the design described in Section 3.4.1, where the CUSTOMER table is horizontally partitioned by its C_W_ID column:

```
SELECT C_ID, C_FIRST, C_MIDDLE, C_LAST, C_BALANCE
  FROM CUSTOMER WHERE C_W_ID = ? AND C_D_ID = ? AND C_LAST = ?
 ORDER BY C_FIRST;
```

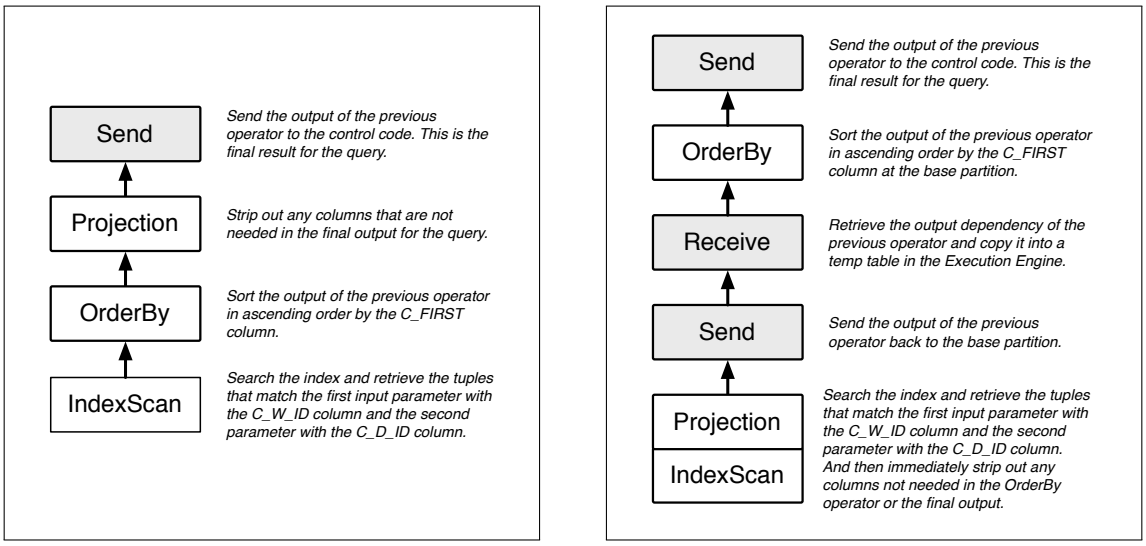
Each invocation of this query is always single-partitioned because its WHERE clause contains an equality predicate on the table's partitioning column. The single-partition plan generated by H-Store's Project Compiler for this query is shown in Fig. 3.5a. The leaf INDEXSCAN operator vertex in the tree retrieves the CUSTOMER records using the index on the C_W_ID and C_D_ID columns. The output table generated by this vertex is passed into the ORDERBY operator. This next operator sorts a copy of its input table based on the values of the C_FIRST column. The output table of this the ORDERBY operator is then passed to the PROJECTION operator to discard any columns from the tuples that are not needed. The final output table is then returned to stored procedure's control code.

The Project Compiler uses a heuristic-based optimizer to refine single-partition query plans. For example, in the above example, the projection operation can be embedded in the scan operator so that it is applied immediately after the DBMS finds the tuple that it matches the query's WHERE clause predicates. This reduces the amount of data that the DBMS needs to copy into the operators' output tables and avoids the overhead of processing an extra operator vertex. The Project Compiler applies this same technique to other operators in single-partition query plans, including LIMIT and DISTINCT qualifiers. Support for operation embedding must be explicitly added to the execution engine and the project compiler needs to be configured as to what operators can be embedded inside of others.

3.5.2 Multi-Partition Query Plans

A multi-partition query plan enables the DBMS to execute a query on multiple partitions and coalesce the results into a single output table [73]. This is not the same as executing a single-partition plan independently at multiple partitions. In some cases, H-Store will execute a portion of a plan at one group of partitions and then transfer their output to execute another part of the plan on another group of partitions [175].

As an example of how H-Store generates a multi-partition plan, consider the same query in the example from above, except now assume that the CUSTOMER table is partitioned by the C_ID column. Since this query does not contain an equality predicate on the C_ID column in its WHERE clause, the DBMS must check every partition in the cluster to find the records that it needs. Each partition will execute this portion of the plan and then sends their intermediate results to the base partition. The execution engine at the base partition waits until it receives responses from each of the other partitions. Once it does, it combines their individual results them into a single intermediate table and performs the final ORDERBY operation to sort the output table. With this approach, all of the global decisions about the query are made at the base partition, while



(a) Single-Partition Query Plan

(b) Multi-Partition Query Plan

Figure 3.5: Examples of a single-partition and multi-partition query plan for the same SQL statement.

each remote partition makes local decisions about what data to return [175]. This is the same strategy used in other distributed DBMSs, such as IBM's R* [225].

The multi-partition query plan is similar to the single-partition plan, except that it contains additional operators to send data between partitions and the plan is divided into multiple fragments. As shown in Fig. 3.5b, the Project Compiler adds these special SEND and RECEIVE operators in a multi-partition query plan to instruct one partition to send an intermediate result to another partition. In the current version of H-Store, these operators can send data from the transaction's base partition to a remote partition or from a remote partition to the base partition. Some commercial distributed DBMSs, such as Clustrix [1], are able to send data directly between partitions during query execution to improve data locality. This is typically only useful for queries that perform a join on many tables, which are more common in OLAP workloads.

H-Store's Project Compiler supports several optimizations for multi-partition plans to reduce the amount of data that one partition will need to send to another [175]. These include pushing LIMIT, DISTINCT, PROJECTION, and AGGREGATE operators down to the lower parts of the query plan tree so that they are executed on the remote partitions. Some operators may have to be executed again at the base partition (e.g., DISTINCT) after the intermediate results from multiple partitions are collected together. But this additional computational cost is not problematic if the optimization reduces the amount of data that is transmitted over the network each time. In the case of non-commutative aggregates (e.g., AVERAGE), the query plan is rewritten so that each partition computes the local sum and count aggregates and then this the base partition computes the weighted average from this data.

3.6 Runtime Operation

We now present the execution process for transactions in H-Store from beginning to end. We begin with describing how H-Store determines what node to invoke a transaction on and how the system decides whether to execute it as a fast, single-partition transaction or a slow, distributed transaction. We then describe how the system routes query invocation requests for running transactions to the partitions with the data that they need to access or modify, and how the execution engines process the plans for these queries and return their results. Finally, we discuss H-Store's commit protocol that ensures that the database stays consistent across multiple nodes.

3.6.1 Transaction Initialization

When a new transaction request arrives at a node, the coordinator needs to determine (1) the partition to invoke the control code for that transaction (i.e., its base partition), and (2) the partition(s) that the transaction is expected to access data from. For the former, a transaction's base partition is computed from the procedure's routing parameter (cf. Section 3.4.4). If the transaction's base partition is not at the node that the request arrived on, then it is redirected to the proper location. When the transaction completes, the result is sent back to the client through the original node. H-Store's client library can be configured to be aware of the database's partitioning scheme so that the application always sends requests to the correct node to avoid incurring these extra network hops.

H-Store supports a variety of methods for determining the partitions that the transaction will access. The default option is to assume that the transaction's base partition is the only partition that the transaction will access and thus all transactions are assumed to be single-partitioned. In VoltDB [10], the developer annotates the stored procedure's control code to statically declare whether the transaction is always single-partitioned or not. Both of these approaches are only approximations and can fail to capture certain nuances for procedures that are only single-partitioned some of the time [180]. The transaction may need to access more partitions than was originally predicted, in which case the transaction will be restarted when it attempts to execute a query that accesses data at a partition that the transaction did not acquire its lock. Or the transaction may only access a subset of the partitions that were identified for it when it was initialized, which means that these partitions are idle while the transaction executes. In the case of VoltDB, if a procedure is marked as always being multi-partition, then the DBMS will lock the entire cluster for each transaction for that procedure. One could allow the application to provide hints to the DBMS, but this would require the developer to embed additional logic that was aware of the physical layout of the database. For the purpose of this discussion, we assume that the DBMS can correctly determine the partitions that the transaction will access. We describe a machine learning-based technique for automatically determining this information at runtime in Chapter 5.

Once the transaction's partitions are computed and the request is at the node that contains the transaction's base partition, then the coordinator assigns the transaction a globally unique id and then queues the request to acquire the locks for the partitions that it needs to access. The transaction starts executing only after it acquires all of these locks.

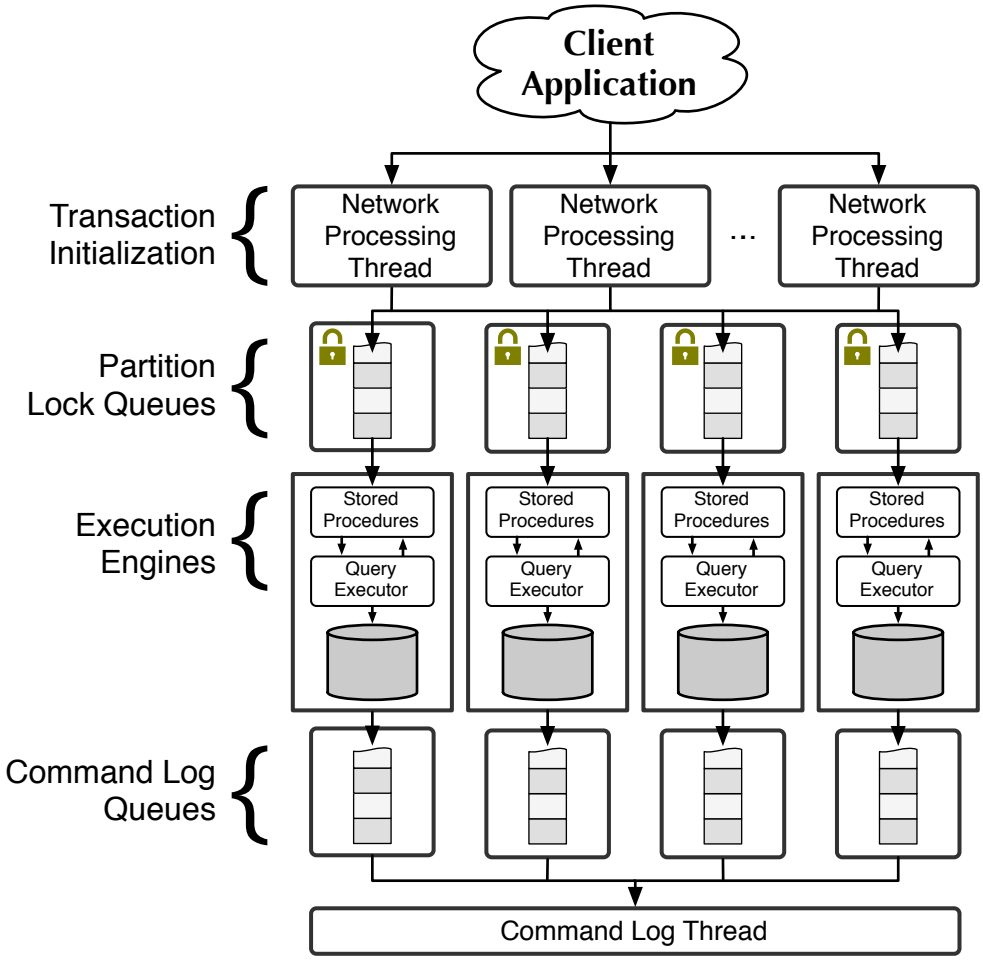


Figure 3.6: The runtime execution flow of a transaction in H-Store.

3.6.2 Query Routing

As a transaction runs, its control code will queue batches of query invocations using H-Store's procedure API and then dispatch them for execution. The control code is blocked until the execution engine finishes executing these queries and returns with their results. For each batch, H-Store uses the *Batch Planner* component to create an *execution schedule* for its queries. This schedule specifies what plans to execute at different partitions for each query invocation and the order in which they are executed. This process, known as *localization* in distributed query processing [175], determines whether a query invocation is executed using a single-partition or multi-partition plan.

H-Store's Batch Planner examines each query invocation in a batch and determines the location of the data that they need to access based on the database's design specification [175, 181]. The plan that the Batch Planner selects for each query invocation in the batch depends on (1) the query's type (i.e., SELECT, INSERT, UPDATE, or DELETE), (2) the partitioning scheme of the table(s) that the query targets, and (3) the query's WHERE clause predicate (for non-INSERT queries). An execution schedule can contain a mixture

of single-partition or multi-partition query plans. The same query could also be invoked multiple times in a batch with some invocations using the single-partition plan and others using the multi-partition plan. If a query invocation will execute on a remote partition and that is the only partition that it accesses, then H-Store still uses the single-partition plan for invocation.

The following rules are how H-Store's Batch Planner determines what plan to use for each query invocation in a batch and how it computes what partitions those queries will access:

1. The simplest case is for an INSERT query on a non-replicated table. With these queries, the Batch Planner examines the values of the input parameters that correspond to that table's partitioning columns to determine what partition to insert the tuple into.

2. A SELECT query that only accesses replicated tables is always executed as a single-partition at the transaction's base partition. If a SELECT query joins a replicated table with a non-replicated table, then the replicated table is ignored when determining what partitions to execute the query and the Batch Planner uses the same process described above.

3. Any INSERT, UPDATE, or DELETE query on a replicated table is always broadcast to all of the partitions in the cluster. This is necessary to ensure that each copy of the table is synchronized.

4. For SELECT, UPDATE, and DELETE queries on non-replicated tables, the Batch Planner examines the query's WHERE clause to determine what partitions it needs to access. It first selects the single-partition plan for each query and computes what partitions it accesses. If the Batch Planner determines that the query needs to access multiple partitions, then it re-computes the partitions using the query's multi-partition plan.

For the last case, the Batch Planner extracts the predicates from the query's WHERE clause to find the predicates on the partitioning column(s) for any non-replicated table (as defined in the application's database design specification). A predicate is represented as a pair where the first element is partitioning column and the second element is either (1) a query input parameter, (2) a constant value, or (3) another column from the same table or a different table. If second element is an input parameter or a constant value, the Batch Planner uses H-Store's internal API to determine what partition the predicate references based on this value. But if the second element of a predicate pair is another column, then the Batch Planner recursively searches the other predicates to find another predicate pair that references an input parameter or a constant value.

As an example of this process, consider the following query in the `StockLevel` transaction from the TPC-C benchmark [216]. Assume that the `ORDER_LINE` and `STOCK` tables are partitioned on their `OL_W_ID` and `S_W_ID` columns, respectively:

```
SELECT COUNT(DISTINCT(OL_I_ID))
  FROM ORDER_LINE, STOCK
 WHERE OL_W_ID = ? AND OL_D_ID = ? AND OL_O_ID < ? AND OL_O_ID >= ?
   AND S_W_ID = OL_W_ID AND S_I_ID = OL_I_ID AND S_QUANTITY < ?
```

For this query, the Batch Planner computes the partitions needed for the `ORDER_LINE` by examining the value of the input parameter used in the predicate on the table's warehouse id (e.g., `OL_W_ID = ?`). But for the

STOCK table, the predicate on its partitioning column references another column (e.g., S_W_ID = OL_W_ID). Since the OL_W_ID column is used in another equality predicate with an input parameter, the Batch Planner replaces that column reference with value of the input parameter used in the other warehouse id predicate (e.g., OL_W_ID = ?). The other predicates are ignored, since they do not contain a partitioning column for either table.

3.6.3 Query Execution

After the Batch Planner generates the execution schedule, the system then distributes the query plans for execution. There are three different execution scenarios in this process:

1. If all of the queries in the batch only access data from the transaction's base partition, then the engine executes the queries immediately and returns the results to the control code.

2. If one or more queries need to access data from a remote partition and the transaction has the locks for those partitions, then the system transmits the query plan execution requests to the appropriate locations through its coordinator.

3. If at least one query in the batch attempts to access data from a remote partition that the transaction does not have the lock for, then the DBMS will abort that transaction and reschedule it to acquire the original locks that it had along with the locks for the partitions that it attempted to access.

When the Batch Planner chooses the multi-partition plan for a query invocation, it groups that plan's operators into fragments based on the partitions that they access. Each fragment has an output dependency and an optional input dependency. The input dependency corresponds to the output table generated by the last operator in the previous plan fragment (if it exists). Likewise, the output dependency is the output table for the last operator in that fragment. The Batch Planner assigns each separate identifiers for a fragment's input and output dependencies. The DBMS uses these identifiers to ensure that it does not schedule a fragment for execution until the other fragments that generate the data that it needs as its input complete.

H-Store executes all queries with an in-memory undo log at each partition so that it can rollback any changes in case that the transaction aborts. Each entry in the undo log contains an ordered list of actions that reverse the modifications made by the transaction's queries since the last log entry. For example, if a transaction executes a query that inserts a new tuple, then the undo action for that query will remove that tuple from the table and any affected indexes. Likewise, for DELETE queries, the removed tuple is added back into the table and its indexes. For updates, the DBMS overwrites a tuple's original fields with the new values, thus the log entry must contain a copy of the original contents of the tuple so that the modified fields can be replaced in the undo action. The DBMS does not copy non-inlined fields (e.g., fields that are not stored directly in the tuple memory space) in the undo action if they were not modified by the update operation.

For each new query batch, the execution engines at each partition decide whether to create a new log entry for that batch. If it is the first query batch for the transaction, then the DBMS will always start a new undo log entry. In the case of single-partition transactions, this will be the only log entry needed for that transaction at its base partition because the transaction will never be pre-preempted by another transaction. For distributed transactions, the engines at all of the partition's involved in the transaction will create a new log entry any

time that the transaction submits a batch that contains queries that will execute on remote partitions. This is necessary to support speculative transaction execution. For example, when the distributed transaction is stalled waiting for query results from remote partitions, the execution engine for the transaction's base partition can execute queued single-partition transactions while it waits [131]. Each of these speculative transactions will create their own undo log entry in case the DBMS needs to rollback their changes without affecting the in-flight changes of the current distributed transaction. We discuss speculative execution in more detail in Chapter 6.

3.6.4 Transaction Commit Protocol

A transaction automatically closes when the control code completes or when it invokes the command in H-Store's API to abort. The execution engine at the transaction's base partition then notifies the other partitions that the transaction is finished. This is to ensure that all of the partitions involved in the transaction agree with the final outcome [35, 93, 142, 160]. The protocol that H-Store uses to complete a transaction depends on whether that transaction was single-partitioned or not.

For single partition transactions, the execution engine immediately commits or aborts the transaction when it finishes. The base partition's engine does not need to coordinate with any other partition because they were not involved in the transaction, thus the transaction can commit quickly without waiting for network communications. This is why it is important that most requests in the system are processed as single-partition transactions.

To commit distributed transactions, H-Store uses the two-phase commit (2PC) protocol [93, 160]. When a distributed transaction's control code finishes at its base partition, the engine sends the "prepare" 2PC message to the transaction's remote partitions through their transaction coordinators. The remote partition that receive this message will check whether the transaction is safe to commit,[2] and then returns an acknowledgement response to the base partition's coordinator. Once the base partition's engine receives acknowledgements from all of the transaction's remote partitions, it sends the "finish" 2PC message to have them commit that transaction and begin processing the next transaction in their queues. If engine does not receive these acknowledgements before an administrator-defined timeout or if one of the responses from the remote partitions is negative (e.g., one partition needs to abort the transaction), then the transaction is aborted. As we will describe in Section 6.4.2, there are special cases where the transaction is aborted at one partition, due to errant speculative queries, but the transaction will still commit at all other partitions. If the control code explicitly requests to abort the transaction, then the base partition's engine only sends the "finish" message to the remote partitions to indicate them to revert the transaction's changes.

When a partition commits a transaction, it discards the undo log entries for that transaction. Likewise, when the transaction is aborted at a partition, that partition's execution engine executes the undo actions in the transaction's log entries and then discards them.

Given that H-Store stores all data in main memory and the computational cost of OLTP queries is small, the overhead of running the 2PC protocol over the network is significant compared to the overall execution

[2] H-Store only checks whether a transaction is safe to commit when using the speculative transaction and query pre-fetching optimizations described in Chapter 6. Non-global integrity constraints are checked immediately after a table is modified. H-Store does not currently support enforcing distributed global integrity and referential constraints.

time of transactions. Thus, there are two optimizations that H-Store employs to reduce the overhead of 2PC. The first is that if a transaction was read-only at a particular remote partition, then the transaction is immediately committed at that partition when it receives the prepare message and the DBMS does not need to send the final 2PC message [193]. This reduces the number of network messages and allows the remote partition to begin processing the next transactions in its queue [130].

The second optimization is when H-Store can identify when a transaction is finished with a remote partition (i.e., it will not execute any more queries at that partition), it will send an early "prepare" message to that partition [72, 160]. This allows the execution engine for this remote partition to speculatively execute other transactions while it waits for the "finish" message. If the transaction tries to execute another query at a remote partition that it already declared as being finish with, then that transaction is aborted and restarted. This technique is called the "early prepare" or "unsolicited vote" optimization, and has been shown to improve both latency and throughput in distributed systems [131, 180, 193].

The two main challenges in implementing the early prepare optimization is (1) how the DBMS will know when the transaction is finished with a particular partition and (2) how to notify that partition that the transaction is finished with it. One way to solve the first problem in H-Store is to have the developer annotate a procedure's control code to indicate to the DBMS that it is submitting the last batch of query invocations for the transaction. The DBMS can then set a flag in the invocation request messages that is sent to the partitions to indicate that it is the last batch. When a remote partition receives a request with this flag enabled, then it will execute the query, return the result to the base partition, and then invoke the checking procedures for the transaction.

One problem with this approach is that it is too coarse-grained. That is, it only indicates that the transaction is finished with all partitions; there is no way for the transaction to easily inform the DBMS that it is done with individual partitions at different parts in the control code. As such, this technique is insufficient for procedures that read data from remote partitions in the beginning of the transaction and then only operate on local data for the remainder of the transaction. This is because the control code has no knowledge of what partitions the queries will touch and thus it would require that transactions maintain their own execution state. To overcome this problem, we discuss in Chapter 5 how H-Store uses machine learning techniques to automatically determine when a transaction is finished with each individual partition at any point of the transaction's lifetime.

When the DBMS learns that a transaction is finished with a partition, it needs to decide how to notify that partition that the transaction no longer needs it. As described above, if the transaction is executing a query at a remote partition that is the last one it will execute at that partition, then the DBMS can set a flag in execution request. But the DBMS may be notified that a transaction is finished with a partition after it has already executed the last query at that partition. Thus, it needs to send this notification separately. If the transaction submits a batch that contains at least one query that needs to execute at a remote partition that is one the same node as the partition that the transaction just marked as being finished, then the DBMS piggybacks the prepare notification onto the query invocation request. If the transaction is not executing any queries at the same node, then it will send a separate notification message. The DBMS will not send this notification if the batch contains only single-partition queries that execute at the transaction's base partition.

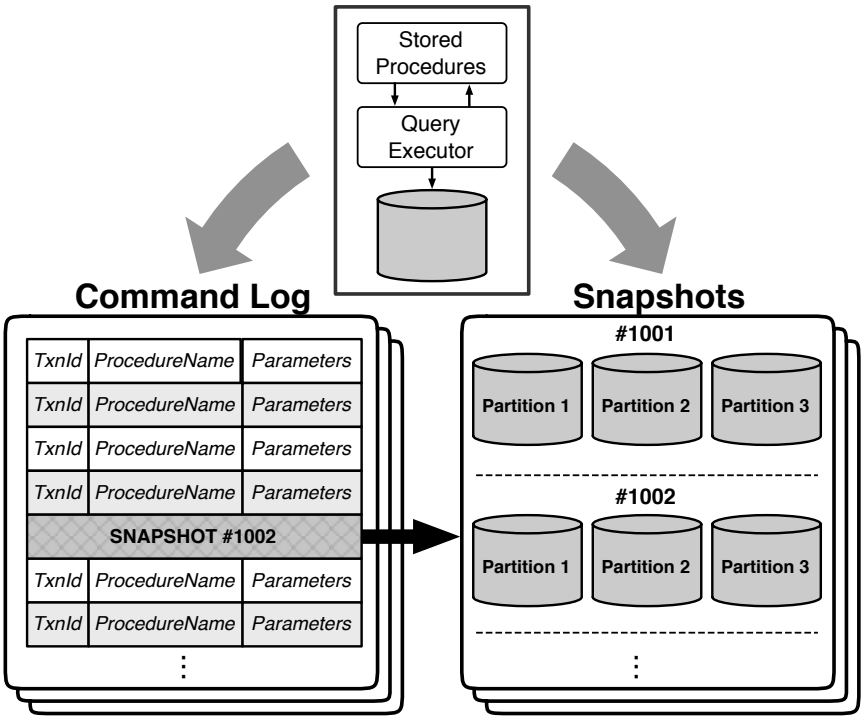


Figure 3.7: An overview of H-Store's logging and checkpoint scheme.

3.7 Recovery Mechanism

Since H-Store is a main memory DBMS, it must ensure that all of a transaction's modifications are durable and are recoverable if a node crashes. The key, however, is to provide this guarantee without significantly hindering the performance advantages of the system. Since H-Store is designed to run on commodity hardware, we cannot assume that the DBMS will be deployed using special-purpose components (e.g., battery-backed up memory) [88].

Given this, H-Store uses a lightweight, logical logging scheme that has less overhead than existing approaches for disk-oriented systems [161]. It will also take periodic checkpoints of the database to reduce the recovery time of the system after a crash. We now discuss these two mechanisms. We then discuss in Section 3.7.3 how H-Store restores the state of the database from these logs and checkpoints.

3.7.1 Command Logging

As described in Section 2.2.3, DBMSs record the changes made by transactions in separate logs on disk. This is to avoid multiple write operations to random locations on the disk to update the tuples modified by each transaction. The log entries for multiple transactions can be written together in a batch to amortize the cost of synchronizing the disk [73, 114].

There are a variety of approaches to transaction logging in DBMSs [106, 115, 128, 155]. At one extreme, there is *physical logging*, where the DBMS records the before and after image of an element in the database

(e.g., a tuple, block, or internal data structure for an index) being modified by the transaction [106, 160]. Another approach is known as *logical logging* (sometimes called *event logging*) where the DBMS only records the high-level operation that the transaction executed (e.g., a query invocation).

Logical logging reduces the amount of data that needs to be written to disk compared to physical logging, since it only needs to record what the operation was rather than what it actually did to the database. But recovering the database using logical logging will take longer because the DBMS will need to re-execute each operation. This is not a significant problem for main memory DBMSs, however, because they are able to execute transactions much faster than disk-oriented systems.

H-Store uses a variant of logical logging, known as *command logging*, where the DBMS only records the transaction invocation requests to the log [155]. Each log record contains the name of the stored procedure and the input parameters sent from the application, and the transaction's id. Because one log record represents the entire invocation of the transaction, command logging does not support transaction save points [160]. This is not a significant limitation because OLTP transactions are short-lived.

H-Store writes out the log records using a separate thread; the execution engines are never blocked by the logging operations. The DBMS writes the entries after the transaction has executed but before it returns the result back to the application. This is different than write-ahead logging [106], where the DBMS logs the transaction request before it executes. This has two advantages. The first is that the DBMS does not need to write out entries for transactions that abort, since the system ensures that all of the changes made by an aborted transaction are rolled back first before executing the next transaction. Thus, an aborted transaction's affect on the database's state is the same as if it was never executed at all. Second, because H-Store can restart transactions due to internal control mechanisms, a transaction may be assigned multiple transaction identifiers. For example, when a transaction attempts to access a partition that it does not have the lock for, the DBMS will restart that transaction, rollback any changes that it made, assign it a new transaction identifier, and then resubmit the lock acquisition requests at the partitions that it needs. If the DBMS logged the transaction's command prior to execution, it would need to write a new entry every time a transaction restarted that marked the previous entry as voided.

The DBMS combines command log entries together for multiple transactions and writes them in a batch to amortize the cost of writing to disk [73, 114, 224]. Modifications made by transactions are not visible to the application until their log record has been flushed. Similarly, a transaction cannot be released to the application until all transactions that executed before it have been written to the command log.

3.7.2 Snapshots

As the DBMS executes transactions and writes their commands out to the log, the DBMS also creates non-blocking snapshots of the database's tables [151, 211]. The snapshot for each partition is written to the local disk at their host node. When the system needs to recover the database after a crash, it loads in the last checkpoint that was created and then replays only the transactions that appear in the command log after this checkpoint [155]. This greatly reduces the time needed to recover the database. H-Store's snapshots only contain the tuples in the tables and not the indexes.

The DBMS can be configured to take checkpoints periodically or manually using a system stored procedure. The system also maintains a catalog of the snapshots that it has taken that is retrievable by the

application through another system procedure.

When H-Store starts a new checkpoint, one node in the DBMS is elected as the coordinating node for the next checkpoint. This node is either selected at random (if it for is a scheduled checkpoint initiated by the DBMS) or the node with the transaction's base partition (if it was initiated through a system procedure). The DBMS at this node sends a special transaction request to every partition in the cluster to instruct them to begin the checkpoint process. This request locks all of the partitions to ensure that each node starts writing the checkpoint from a transactionally consistent database state [182]. This system procedure causes each execution engine at all of the partitions to switch into a special "copy-on-write" mode. With this, any changes made by future transactions do not overwrite the tuples that existed when the current checkpoint started and any new tuples that were inserted after the checkpoint was started are not included in the snapshot data. Once all of the partitions send back acknowledgements to begin the snapshot, the DBMS commits the special transaction and each partition starts writing out the snapshot to disk in a separate thread. The execution engines then return to processing transactions while the snapshot processing occurs in the background.

The amount of time that it takes the execution engine at a a partition to complete the database snapshot out to disk depends on the size of the database and the write speed of the storage device. After a partition's engine finishes writing the snapshot, it disables the "copy-on-write" mode and sends a notification message back to the coordinating node. Once the coordinating node has notifications from every partition, it sends a final finish message to each partition that instructs them to clean up transient data structures and marks the snapshot as complete.

3.7.3 Crash Recovery

The process for restoring a database from the command log and snapshot is straightforward [155]. First, when the H-Store node starts, each execution engine at the node reads in contents from the last snapshot taken at its partition. For each tuple in a snapshot, the DBMS has to determine what partition should store that tuple, since it may not be the same one that is reading in that snapshot. This situation can occur if the administrator changes number of partitions in the cluster when the system was brought off-line. As the engines load in each row, it will also rebuild the tables' indexes at its partitions.

Once the snapshot has been loaded into memory from the file on disk, the DBMS will then replay the command log to restore the database to state that it was in before the crash. A separate thread scans the log backwards to find the record that corresponds to the transaction that initiated the snapshot that was just loaded in. It then scans the log in the forward direction from this point and re-submits each entry as a new transaction request at that node. The transaction coordinator ensures that these transactions are executed in the exact order that they arrived in the system; this differs from the normal execution policy where transactions are allowed to get re-ordered and re-executed.

The state of the database after this recovery process is guaranteed to be correct, even if the number of partitions during replay is different from the number of partitions at runtime. This is because (1) transactions are logged and replayed in serial order, so the re-execution occurs in exactly the same order as in the initial execution, and (2) replay begins from a transactionally-consistent snapshot that does not contain any uncommitted data, so no rollback is necessary at recovery time [106, 155, 182].

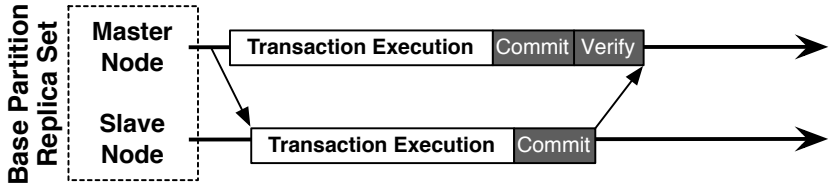


Figure 3.8: A timeline diagram for the execution of a single-partition transaction with one replica. Each node in the replica set will execute the transaction's control code independently. The master node will then verify at the end that all of its slave nodes returned the same result.

3.8 Replication

Because of the potentially long recovery time in H-Store to rebuild the database from the command log and snapshots, a production OLTP deployment will use a replication scheme that allows the system to keep running even if one node crashes. Replication is the standard technique to protect against node failures in distributed systems [97] and reduces the likelihood that the DBMS will need to perform a full recovery [155].

A *replica set* is a collection of nodes that all contain the same partitions. Each node maintains its own command log and database snapshots. The system uses the Paxos consensus protocol to elect one node as the *master* (i.e., initiator) of the replica set [141]. The other nodes in the corresponding replica set are referred to as *slave* nodes. When the master node fails, the system runs a new election round to promote one of the slaves as the new master. The DBMS remains on-line as long as k replicas nodes are still operational for each replica set, where k is an administrator-defined failure threshold [210]. Hence, in terms of the CAP theorem [91], H-Store is strongly consistent and highly available, but it is not network partition-tolerant.

We now discuss how H-Store executes single-partition and distributed transactions on deployments with replicated nodes.

3.8.1 Single-Partition Transactions

When the master node receives a new single-partition transaction request, it sends it to the slave nodes in its replica set. Each node in the replica set will execute the control code for the transaction's procedure independently. They do not need to coordinate the queries that the transaction invokes while it executes. This is another reason why the stored procedure's control code must be deterministic (cf. Section 3.3). Since the transaction is independently executed at each replica, the database transformation must be the same at each node to ensure that the final state of the database after the transaction commits is the same.

The master node waits until it gets responses from each of the replicas. It then verifies that all of the replicas produced the same result for that transaction. As long as a majority of the transactions agree, then the result is sent back to the application. Any replica whose response deviates from the agreed upon result is considered to have a hardware failure and the node is shut down.

3.8.2 Distributed Transactions

Since distributed transactions could modify multiple partitions, H-Store cannot use the same execution strategy for these transactions as it uses for single-partition transactions. Otherwise, a transaction that invokes

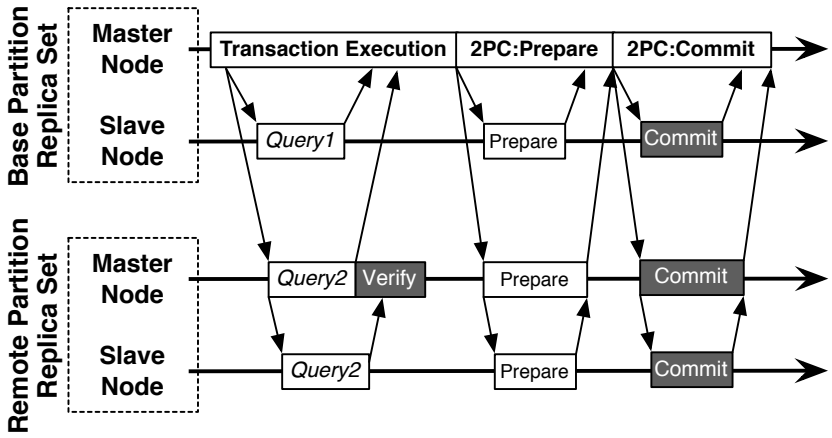


Figure 3.9: A timeline diagram for H-Store's replication scheme when executing a distributed transaction. The transaction needs to execute one query at its base partition (i.e., `Query1`) and a different query at its remote partition (i.e., `Query2`). Each partition is replicated on two nodes.

queries that modify remote partitions would have those queries executed multiple times (one for each replica). Thus, instead of executing the transaction's control code at every node in the its replica set, the control code will only execute at the master node's base partition. As the transaction invokes queries on remote partitions, they are broadcast to the replicas. The master node does not need to wait for each replica to acknowledgement that it completed each query successfully. It only needs to confirm that they executed the all of the queries for the transaction in the correct order during the prepare phase in 2PC. If the replicas fail to agree with the master's execution order, then the transaction is aborted.

3.9 Comparison

We now revisit the Voter benchmark used in the evaluation in Section 2.1. This time we ran the same experiment with H-Store using its built-in benchmark framework (cf. Appendix A). We first deployed the DBMS with one partition to process transactions (i.e., one CPU core), and then ran the benchmark three times we ran the benchmark three times and report the average results. Just as with MySQL and Postgres, for each trial run we allowed the DBMS to "warm-up" for 60 seconds and then measured the number of transactions completed after five minutes. We then increased the number of partitions allocated in the DBMS. We configured H-Store with a design specification that makes all Voter transactions single-partitioned. The system also uses command logging to write out transactions entries to disk to provide the same persistence and durability guarantees as MySQL and Postgres.

The results in Fig. 3.10 show that with only a single partition, H-Store processes $\sim$25,000 transactions per second. This is over twice the maximum throughput of both of the traditional DBMSs. This highlights the advantages of H-Store's serial execution model; transactions never stall waiting to acquire a lock held by another transaction. As more cores are allocated to it, H-Store is able to almost able to scale linearly. At eight partitions, H-Store's throughput is $\sim$250,000 transactions per second. This is over 25$\times$ the throughput achieved by either MySQL or Postgres.

The inherent limitation of main memory DBMSs is that they can only support that are smaller than the

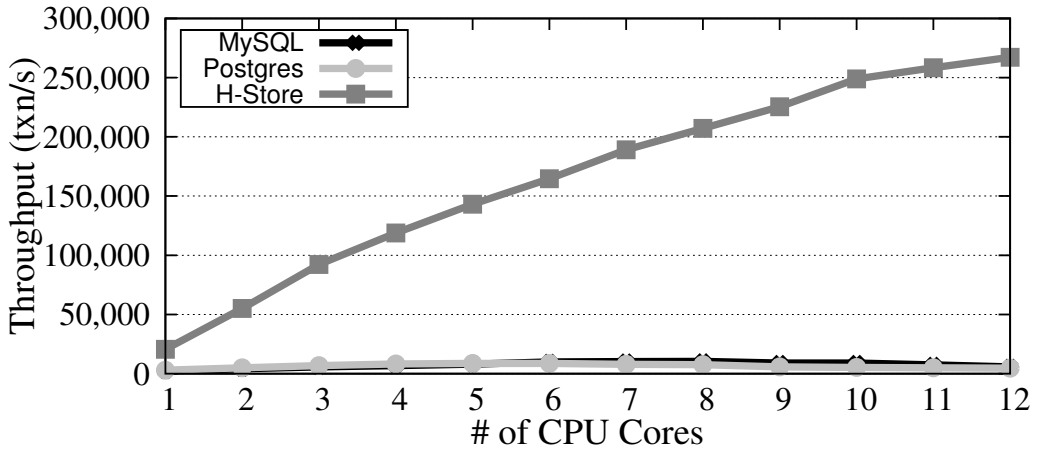


Figure 3.10: The single-node throughput measurements for the Voter benchmark on MySQL, Postgres, and H-Store. All DBMSs were deployed with the same serializable isolation level and durability guarantees.

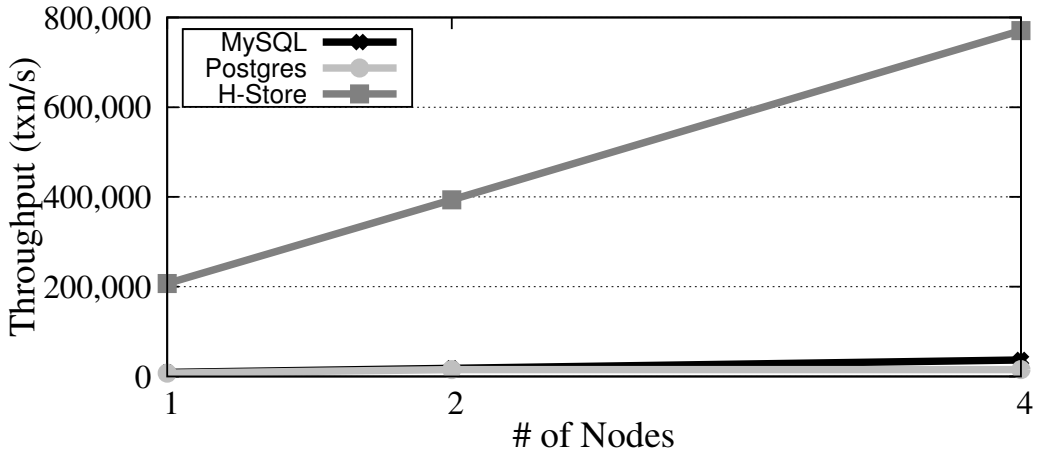


Figure 3.11: The multi-node throughput measurements for the Voter benchmark on MySQL, Postgres, and H-Store. Each cluster configuration has eight CPU cores per node.

amount of memory on a single node. As described above, H-Store overcomes this by supporting multi-node deployments out of the box. We ran the Voter benchmark again, but this time we scaled the system out over one, two, and four nodes (with eight partitions per node). Since all of the transactions in this benchmark are single-partitioned, we had ran multiple instances of the client drivers for MySQL and Postgres trials that were each configured to target one node in the cluster.

Once again, the results in Fig. 3.11 show that H-Store is able to scale almost linearly as the number of nodes in the cluster is doubled. The throughput for MySQL and Postgres also scales linearly, but it is still significantly less than what H-Store achieves with the same hardware.

Chapter 4

Automatic Database Design

The scalability of OLTP applications on many of the distributed DBMSs, such as H-Store, depends on the existence of an optimal *database design* (i.e., partitioning and replication scheme). Such a design defines how an application's data and workload is partitioned or replicated across nodes in a cluster, and how queries and transactions are routed to nodes. This in turn determines the number of transactions that access data stored on each node and how skewed the load is across the cluster. Optimizing these two factors is critical to scaling systems: we collected experimental evidence that a growing fraction of distributed transactions and load skew can degrade performance by over a factor $10\times$. Hence, without a proper design, a DBMS will perform no better than a single-node system due to the overhead caused by blocking, inter-node communication, and load balancing issues [131, 180].

Many of the existing techniques for automatic database partitioning, however, are tailored for large-scale analytical applications (i.e., data warehouses) [179, 195]. These approaches are based on the notion of *data declustering* [153], where the goal is to spread data across nodes so that the execution of each query is parallelized across them [18, 50, 188, 230]. Much of this corpus is not applicable to OLTP systems because the multi-node coordination required to achieve transaction consistency dominates the performance gains obtained by intra-query parallelism; previous work [66] has shown that, even ignoring effects on lock-contention, this overhead can be up to 50% of the total execution time of a transaction when compared to single-node execution.

Although other work has focused on distributed OLTP database design [66, 230], these approaches lack three features that are crucial for enterprise OLTP databases: (1) support for stored procedures to increase execution locality, (2) the use of replicated secondary indexes to reduce distributed transactions, and (3) handling of time-varying skew in data accesses to increase cluster load balance. These three salient aspects of enterprise databases hinder the applicability and effectiveness of the previous work.

Given the lack of an existing solution for our problem domain, we present *Horticulture,* a scalable tool to automatically generate database designs for stored procedure-based distributed OLTP systems that is focused on complex schemas and that can handle temporally-skewed workloads. The two key contributions in this work are (1) an automatic database partitioning algorithm based on an adaptation of the *large-neighborhood search* technique [86] and (2) a new analytical cost model that estimates the coordination cost and load

distribution for a sample workload. Horticulture analyzes a database schema, the structure of the application stored procedures and sample transaction workload, and then automatically generates partitioning strategies that minimize distribution overhead, while balancing access skew. The run time of this analysis is independent of database size, and thus is not subject to the scalability limits of existing solutions [66, 230]. Moreover, the partitioning strategies that Horticulture supports are not limited to horizontal partitioning and replication but also include replication of secondary indexes, and stored procedure routing.

Although our tool produces database designs that are usable with any shared-nothing DBMS or middleware solution, we integrated and tested Horticulture with H-Store. Testing on main memory DBMSs like H-Store presents an excellent challenge for Horticulture because they are particularly sensitive to the quality of partitioning in the database design, and require a large number of partitions.

We thoroughly validated the quality of our partitioning by comparing Horticulture with four competing approaches, including another state-of-the-art database design tool [66], and running several experiments on five enterprise-class OLTP benchmarks: TATP, TPC-C (standard and skewed), TPC-E, SEATS, and AuctionMark. Our tests show that the three novel contributions of our system (i.e., stored procedure routing, replicated secondary indexes, and temporal-skew management) are much needed in the context of enterprise OLTP systems. Furthermore, our results indicate that our design choices provide an overall performance increase of up to a factor $4\times$ against a state-of-the-art partitioning tool [66] and up to a factor $16\times$ against a practical baseline approach.

4.1 Distributed Transaction Overhead

To illustrate how the presence of distributed transactions affects performance, we executed a workload derived from the TPC-C benchmark [216] on H-Store. We postpone the details of our experimental setting to Section 4.6. In each round of this experiment, we varied the number of distributed transactions and execute the workload on five different cluster sizes, with at most seven partitions assigned per node. Fig. 4.1a shows that a workload mix of just 10% distributed transactions has a significant impact on throughput. The graph shows that the performance difference increases with larger cluster sizes: at 64 partitions, the impact is approximately $2\times$. This is because single-partition transactions in H-Store execute to completion in a single thread, and thus do not incur the overhead of traditional concurrency control schemes [112]. For the distributed transactions, the DBMS's throughput is limited by the rate at which nodes send and receive the two-phase commit messages. These results also show that the performance repercussions of distributed transactions increases relative to the number of partitions because the system must wait for messages from more nodes. Therefore, a design that minimizes both the number of distributed transactions and the number of partitions accessed per transaction will reduce coordination overhead, thereby increasing the DBMS's throughput [66, 230].

Related to this, determining the "best" base partition for each stored procedure subject to the tables' layout will also improve performance because it reduces the number of transactions that must be aborted and restarted. This is the approach used by IBM DB2 [59]: if a transaction deemed single-partitioned executing on one node attempts to access data on another, then the DBMS will abort that transaction and restart it at the other location. But this technique has been shown not to scale as the number of partitions increase, since

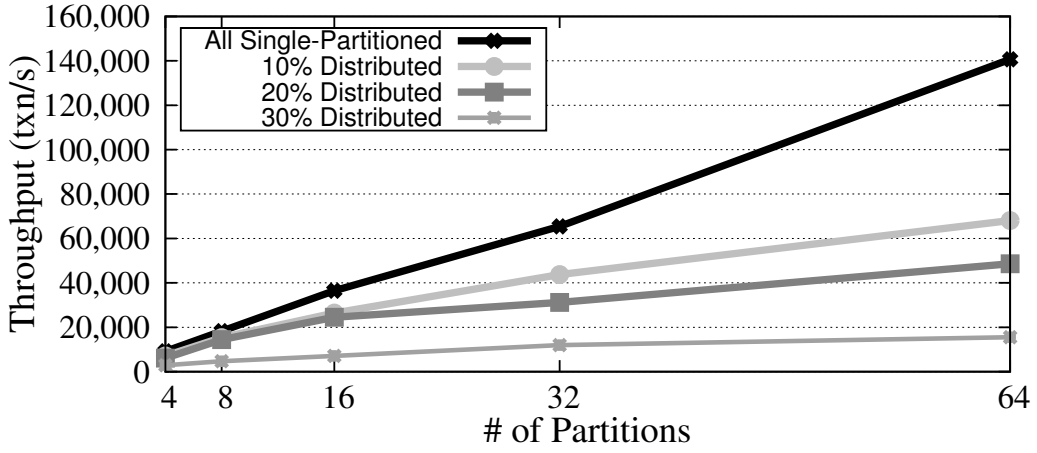
the likelihood that a transaction is already at the best base partition decreases as the number of partitions increases. As an alternative, a database design can designate one or more attributes for each procedure that will indicate the base partition for each new request at run time (e.g., by hashing the value of that attribute). This allows the DBMS to quickly determine the best base partition for a transaction, assuming that the procedure is always single-partitioned.

Even if a given database design enables every transaction to execute as single-partitioned, the DBMS may still fail to scale linearly if the application's workload is unevenly distributed across the nodes. Thus, one must also consider the amount of data and transactions assigned to each partition when generating a new database design, even if certain design choices that mitigate skew cause some transactions to be no longer singled-partitioned. Existing techniques have focused on *static skew* in the database [66, 230], but failed to consider *temporal skew* [223]. Temporally skewed workloads might appear to be uniformly distributed when measured globally, but can have a significant effect on not only performance but also availability in shared-nothing DBMSs [87].

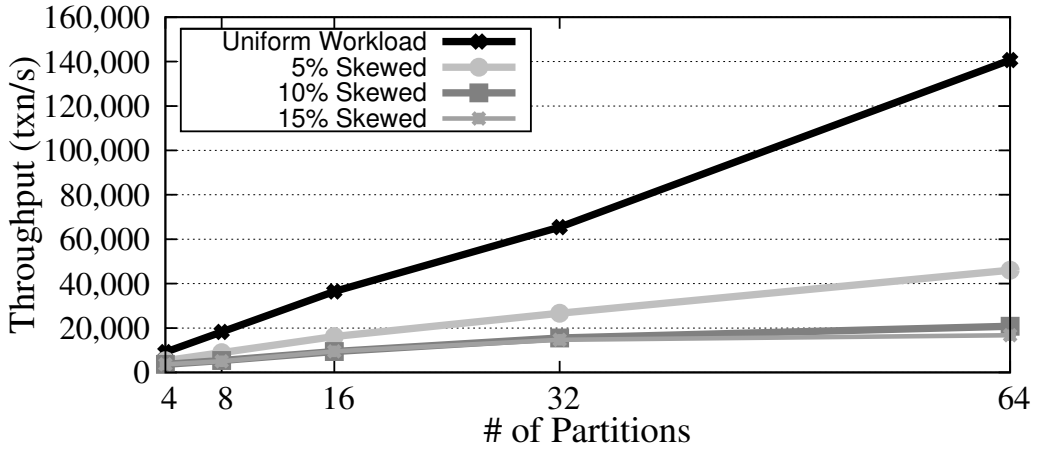
As a practical example of temporal skew, consider Wikipedia's approach to partitioning its database by language (e.g., English, German) [11]. This strategy minimizes the number of distributed transactions since none of the common transactions access data from multiple languages. This might appear to be a reasonable partitioning approach, however the database suffers from a non-trivial amount of temporal skew due to the strong correlation between languages and geographical regions: the nodes storing the articles for one language are mostly idle when it is night time in the part of the world that speaks that language. If the data set for a particular language is large, then it cannot be co-located with another partition for articles that are mostly accessed by users from another part of the world. At any point during the day the load across the cluster is significantly unbalanced even though the average load of the cluster for the entire day is uniform. Wikipedia's current solution is to over-provision nodes enough to mitigate the skew effects, but a temporal-skew-aware database design may achieve identical performance with lower hardware and energy costs.

We also experimentally tested the impact of temporal skew on our H-Store cluster. In this experiment, we use a 100% single-partition transaction workload (to exclude distribution costs from the results) and impose a time-varying skew. At fixed time intervals, a higher percentage of the overall workload is directed to one partition in the cluster. The results are shown in Fig. 4.1b. For large number of partitions, even when only an extra 5% of the overall load is skewed towards a single-partition, the throughput is reduced by a large factor, more than $3\times$ in our test. This is because the execution engine for the partition that is receiving a larger share of the workload is saturated, which causes other partitions to remain idle while the clients are blocked waiting for results. The latency increases further over time since the target partition cannot keep up with the increased load.

The above examples show that both distributed transactions and temporal workload skew must be taken into account when deploying a distributed database in order to maximize its performance. Manually devising optimal database designs for an arbitrary OLTP application is non-trivial because of the complex trade-offs between distribution and skew: one can enable all requests to execute as single-partitioned transactions if the database is put on a single node (assuming there is sufficient storage), but one can completely remove skew if all requests are executed as distributed transactions that access data at every partition. Hence, a tool is needed



(a) Distributed Transactions



(b) Workload Skew

Figure 4.1: The impact on H-Store's throughput for different workload variations in TPC-C `NewOrder`.

that is capable of partitioning enterprise OLTP databases to balance these conflicting goals.

4.2 Database Design Challenges

Horticulture is an automatic database design tool that selects the best physical layout for a distributed DBMS that minimizes the number of distributed transactions while also reducing the effects of temporal skew. As described in Section 3.4, a database design specification for a distributed DBMS defines:

1. The horizontal partitioning attribute(s) for tables,
2. Whether a table is replicated at every partition,
3. Any secondary indexes that are replicated at every partition,
4. The routing attribute(s) for stored procedures.

The administrator provides Horticulture with (1) the database schema of the target OLTP application, (2) a set of stored procedures definitions, and (3) a reference workload trace. A workload trace is a log of previously executed transactions for an application. Each transaction record in the trace contains its procedure input parameters, the timestamps of when it started and finished, and the queries it executed with their corresponding input parameters. Horticulture works under the reasonable assumption that the sample trace is representative of the target application.

Using these inputs, Horticulture explores an application's solution space, where for each table the tool selects whether to (1) horizontally partition or (2) replicate on all partitions, as well as to (3) replicate a secondary index for a subset of its columns. The DBMS uses the column(s) selected in these design elements with either hash or range partitioning to determine at run time which partition stores a tuple. The tool also needs to determine how to enable the DBMS to effectively route incoming transaction requests to the partition that has most of the data that each transaction will need to access [169]. As we will discuss in this section, this last step is particularly challenging for applications that use stored procedures.

The problem of finding an optimal database design is known to be NP-Complete [164, 176], and thus it is not practical to examine every possible design to discover the optimal solution [230]. Even if one can prune a significant number of the sub-optimal designs by discarding unimportant table columns, the problem is still exceedingly difficult when one also includes stored procedure routing parameters—as a reference, the number of possible solutions for TPC-C and TPC-E are larger than 10^{66} and 10^{94}, respectively. Indeed, we initially developed an iterative greedy algorithm similar to the one proposed in [16], but found that it obtained poor results for these complex instances because it is unable to escape local minima. There are, however, existing search techniques from optimization research that make problems such as this more tractable.

Horticulture employs one such approach, called *large-neighborhood search* (LNS), to explore potential designs off-line in a guided manner [68, 86]. LNS compares potential solutions with a *cost model* that estimates how well the DBMS will perform using a particular design for the sample workload trace without needing to actually deploy the database. For this work, we use a cost model that seeks to optimize throughput by minimizing the number of distributed transactions [66, 90, 159] and the amount of access skew across servers [223]. Since the cost model is separate from the search model, one could replace it to generate designs that accentuate other aspects of the database (e.g., minimizing disk seeks, improving crash resiliency). We discuss alternative cost models for Horticulture for other DBMSs in Section 8.3.

We now present our LNS-based approach in the next section, and then describe in Section 4.4 how Horticulture estimates the number of distributed transactions and the amount of skew for each design. Various optimization techniques, such as how to extract, analyze, and compress information from a sample workload trace efficiently and to speed up the search time, are discussed in Section 4.5.

4.3 Large-Neighborhood Search

LNS is well-suited for our problem domain because it explores large solution spaces with a lower chance of getting caught in a local minimum and has been shown to converge to near-optimal solutions in a reasonable amount of time [86]. An outline of Horticulture's design algorithm is as follows:

1. Analyze the sample workload trace to pre-compute information used to guide the search process.

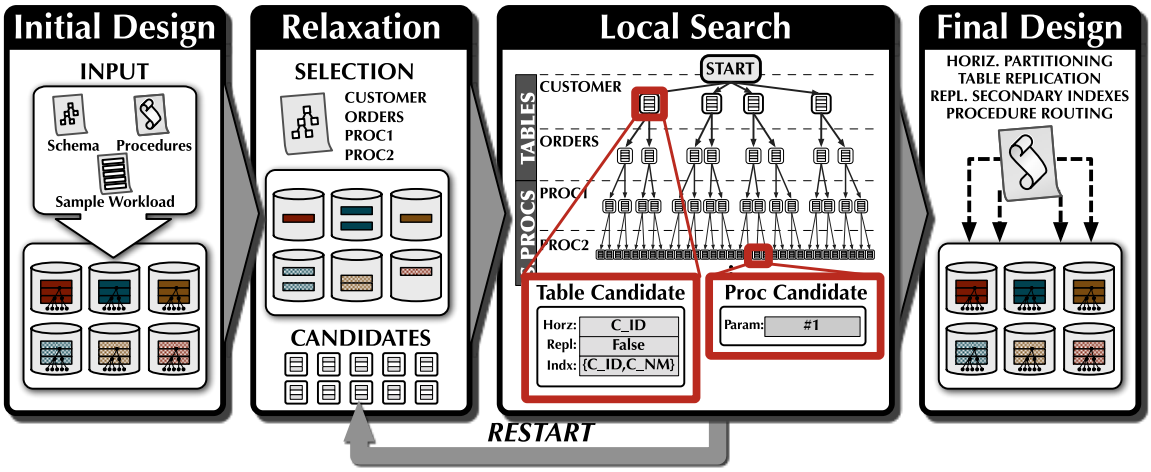


Figure 4.2: An overview of Horticulture's LNS design algorithm. The algorithm generates a relaxed design from the initial design and then uses local search to explore solutions. Each level of the search tree contains the different candidate attributes for tables and procedures for the target database. After the search finishes, the process either restarts or emits the best solution found.

2. Generate an initial "best" design $\mathcal{D}_{best}$ based on the database's most frequently accessed columns.

3. Create a new incomplete design $\mathcal{D}_{relax}$ by "relaxing" (i.e., resetting) a subset of $\mathcal{D}_{best}$.

4. Perform a local search [230] for a new design using $\mathcal{D}_{relax}$ as a starting point. If any new design has a lower cost than $\mathcal{D}_{best}$, then mark it as the new $\mathcal{D}_{best}$. The search stops when a certain number of designs fail to improve on $\mathcal{D}_{best}$ or there are no designs remaining in $\mathcal{D}_{relax}$'s neighborhood.

5. If the total time spent thus far exceeds a limit, then halt the algorithm and return $\mathcal{D}_{best}$. Otherwise, repeat Step 3 for a new $\mathcal{D}_{relax}$ derived from $\mathcal{D}_{best}$.

When generating either the initial design in Step 1 or subsequent designs using local search in Step 4, Horticulture verifies whether a design is *feasible* for the target cluster (i.e., the total size of the data stored on each node is less than its storage limit) [68]. Non-feasible designs are immediately discarded.

Next, we describe each of these steps in more detail.

4.3.1 Initial Design

The ideal initial design is one that is easy to compute and provides a good upper bound to the optimal solution. This allows LNS to discard many potential designs at the beginning of the search because they do not improve on this initial design. To this purpose our system builds compact summaries of the frequencies of access and co-access of tables, called *access graphs*. We postpone the detailed discussion of access graphs and how we derive them from a workload trace to Section 4.5.1.

Horticulture uses these access graphs in a four-part heuristic to generate an initial design:

1. Select the most frequently accessed column in the workload as the horizontal partitioning attribute for each table.

2. Greedily replicate read-only tables if they fit within the partitions' storage space limit.

3. Select the next most frequently accessed, read-only column in the workload as the secondary index attribute for each table if they fit within the partitions' storage space limit.

4. Select the routing parameter for stored procedures based on how often the parameters are referenced in queries that use the table partitioning columns selected in Step 1.

To identify which read-only tables in the database to replicate in Step 2, we first sort them in decreasing order by each table's *temperature* (i.e., the size of the table divided by the number of transactions that access the table) [61]. We examine each table one-by-one according to this sort order and calculate the new storage size of the partitions if that table was replicated. If this size is still less than the amount of storage available for each partition, then we mark the table as replicated. We repeat this process until either all read-only tables are replicated or there is no more space.

We next select the secondary index column for any non-replicated table as the one that is both read-only and accessed the most often in queries' predicates that do not also reference that table's horizontal partitioning column chosen in Step 1. If this column generates an index that is too large, we examine the next most frequently access column for the table.

Now with every table either replicated or partitioned in the initial design, Horticulture generates *parameter mappings* [181] from the workload trace that identify (1) the procedure input parameters that are also used as query input parameters and (2) the input parameters for one query that are also used as the input parameters for other queries. These mappings allow Horticulture to identify without using static code analysis which queries are always executed with the same input parameters using the actual values of the input parameters in the workload. The technique described in [181] removes spurious results for queries that reference the same columns but with different values. We then select a routing attribute for each stored procedure as the one that is mapped to the queries that are executed the most often with predicates on the tables' partitioning columns. If no sufficient mapping exists for a procedure, then its routing attribute is chosen at random.

4.3.2 Relaxation

Relaxation is the process of selecting random tables in the database and resetting their chosen partitioning attributes in the current best design. The partitioning option for a relaxed table is undefined in the design, and thus the design is *incomplete*. We discuss how to calculate cost estimates for incomplete designs in Section 4.4.3.

In essence, relaxation allows LNS to escape a local minimum and to jump to a new neighborhood of potential solutions. This is advantageous over other approaches, such as tableau search, because it is relatively easy to compute and does not require the algorithm to maintain state between relaxation rounds [86]. To generate a new relaxed design, Horticulture must decide (1) how many tables to relax, (2) which tables to relax, and (3) what design options will be examined for each relaxed table in the local search.

As put forth in the original LNS papers [68, 86], the number of relaxed variables (i.e., tables) is based on how much search time remains as defined by the administrator. Initially, this size is 25% of the total number

of tables in the database; as time elapses, the limit increases up to 50%[1]. Increasing the number of tables relaxed over time in this manner is predicated on the idea that a tighter upper bound will be found more quickly if the initial search rounds use a smaller number of tables, thereby allowing larger portions of the solution space to be discarded in later rounds [68, 86].

After computing the number of tables to reset, Horticulture then randomly chooses which ones it will relax. If a table is chosen for relaxation, then all of the routing parameters for any stored procedure that references that table are also relaxed. The probability that a table will be relaxed in a given round is based on their temperatures [61]: a table that is accessed frequently more likely to be selected to help the search find a good upper bound more quickly [86]. We also reduce these weights for small, read-only tables that are already replicated in the best design. These are usually the "look-up" tables in OLTP applications [211], and thus we want to avoid exploring neighborhoods where they are not replicated.

In the last step, Horticulture generates the *candidate attributes* for the relaxed tables and procedures. For each table, its candidate attributes are the unique combination of the different design options available for that table. For example, one potential candidate for `CUSTOMER` table is to horizontally partition the table on the customer's name, while another candidate partitions the table on the customer's id and includes a replicated secondary index on the customer id and name. Multiple candidate attributes for a single table are grouped together as an indivisible "virtual" attribute. The different options in one of these virtual attributes are applied to a design all at once so that the estimated cost never decreases during the local search process.

4.3.3 Local Search

Using the relaxed design $\mathcal{D}_{relax}$ produced in the previous step, Horticulture executes a two-phase search algorithm to iteratively explore solutions. This process is represented as a search tree, where each *level* of the tree coincides with one of the relaxed database elements. As shown in Fig. 4.2, the search tree's levels are split into two sections corresponding to the two search phases. In the first phase, Horticulture explores the tables' candidate attributes using a branch-and-bound search [166, 230]. Once all of the relaxed tables are assigned an attribute in $\mathcal{D}_{relax}$, Horticulture then performs a brute-force search in the second phase to select the stored procedures' routing parameters.

As Horticulture explores the table portion of the search tree, it changes the current table's design option in $\mathcal{D}_{relax}$ to each candidate attribute and then estimates the cost of executing the sample workload using that new design. If this cost estimate is less than the cost of $\mathcal{D}_{best}$ and is feasible, then the search traverses down the tree and examines the next table's candidate attributes. But if this cost is greater than or equal to the cost of $\mathcal{D}_{best}$ or if the design is not feasible, the search continues on to the next candidate attribute for the current table. If there are no more attributes for this level, then the search "backtracks" to the previous level.

Horticulture maintains counters for backtracks and the amount of time spent in the current search round. Once either of these exceed a dynamic limit, the local search halts and returns to the relaxation step. The number of backtracks and search time allowed for each round is based on the number of tables that were relaxed in $\mathcal{D}_{relax}$. As these limits increases over time, the search is given more time to explore larger neighborhoods. We explore the sensitivity of these parameters in our evaluation in Section 4.6.6.

[1] These values were empirically evaluated following standard practice guidelines [68].

In the second phase, Horticulture uses a different search technique for procedures because their design options are independent from each other (i.e., the routing parameter for one procedure does not affect whether other procedures are routed correctly). Therefore, for each procedure, we calculate the estimated costs of its candidate attributes one at a time and then choose the one with the lowest cost before moving down to the next level in the search tree. We examine the procedures in descending order of invocation frequency so that the effects of a bad design are discovered earlier.

If Horticulture reaches the last level in the tree and has a design that is both feasible and has a cost that is less than $\mathcal{D}_{best}$, then the current design becomes the new best design. The local search still continues but now all comparisons are conducted with the new lower cost. Once either of the search limits is reached or when all of the tree is explored, the process restarts using a new relaxation.

The entire process halts after after an administrator-defined time limit or when Horticulture fails to find a better design after a certain period of time (Section 4.6.6). The final output is the best design found overall for the application's database. The administrator then configures the DBMS using the appropriate interface to deploy their database according to this design. We leave the problem of how to reorganize an already deployed database as future work, but note that the high-level process described in this section is likely to remain the same.

4.4 Temporal Skew-Aware Cost Model

Horticulture's LNS algorithm relies on a cost model that can estimate the cost of executing the sample workload using a particular design [61, 157, 176, 230]. Using an analytical cost model is an established technique in automatic database design and optimization [56, 79], as it allows one to determine whether one design choice is better than others and can guide the search process towards a solution that accentuates the properties that are important in a database. But it is imperative that these estimations are computed quickly, since the LNS algorithm can generate thousands of designs during the search process. The cost model must also be able to estimate the cost of an incomplete design. Furthermore, as the search process continues down the tree, the cost estimates must increase monotonically as more variables are set in an incomplete design. That is, for two incomplete designs $\mathcal{D}$ and $\mathcal{D}'$, where $\mathcal{D} \subset \mathcal{D}'$, and a workload $\mathcal{W}$, we require that $cost(\mathcal{D}, \mathcal{W}) \leq cost(\mathcal{D}', \mathcal{W})$.

Given these requirements, our cost model is predicated on the key observation that the execution overhead of a multi-partition transaction is significantly more than a single-partition transaction [112, 211]. Some OLTP DBMSs execute a single-partition transaction serially on a single node with reduced concurrency control, whereas any distributed transactions must use an expensive concurrency control scheme to coordinate execution across two or more partitions [131, 181, 211]. Thus, we estimate the run time cost of a workload as being proportional to the number of distributed transactions.

In addition to this, we also assume that (1) either the working set for an OLTP application or its entire database is stored in main memory and (2) that the run times for transactions are approximately the same. This means that unlike other existing cost models [56, 61, 230], we can ignore the amount of data accessed by each transaction, and that all of a transaction's operations contribute an equal amount to the overall load of each partition. In our experience, transactions that deviate from these assumptions are likely analytical operations that are either infrequent or better suited for a data warehouse DBMS.

Algorithm 1 $CoordinationCost(\mathcal{D}, \mathcal{W})$

$txnCount \leftarrow 0, dtxnCount \leftarrow 0, partitionCount \leftarrow 0$
for all $txn \in \mathcal{W}$ **do**
  $P \leftarrow GetPartitions(\mathcal{D}, txn)$
  if $|P| > 1$ **then**
    $dtxnCount \leftarrow dtxnCount + 1$
    $partitionCount \leftarrow partitionCount + |P|$
  end if
  $txnCount \leftarrow txnCount + 1$
end for
return $\left(\frac{partitionCount}{(txnCount \times numPartitions)} \times \left(1.0 + \frac{dtxnCount}{txnCount}\right)\right)$

We developed an analytical cost model that not only measures how much of a workload executes as single-partition transactions, but also measures how uniformly load is distributed across the cluster. The final cost estimation of a workload $\mathcal{W}$ for a design $\mathcal{D}$ is shown below as the function *cost*$(\mathcal{D}, \mathcal{W})$, which is the weighted sum of the normalized coordination cost and the skew factor:

$$\textit{cost}(\mathcal{D}, \mathcal{W}) = \frac{(\alpha \times CoordinationCost(\mathcal{D}, \mathcal{W})) + (\beta \times SkewFactor(\mathcal{D}, \mathcal{W}))}{(\alpha + \beta)} \tag{4.1}$$

The parameters α and β can be configured by the administrator. In our setting, we found via linear regression that the values five and one respectively provided the best results. All experiments were run with this parameterization.

This cost model is not intended to estimate actual run times, but rather as a way to compare the quality of competing designs. It is based on the same assumptions used in H-Store's distributed query planner. We show that the underlying principals of our cost model are representative of actual run time performance in Section 4.6.3.

4.4.1 Coordination Cost

We define the function $CoordinationCost(\mathcal{D}, \mathcal{W})$ as the portion of the cost model that calculates how well $\mathcal{D}$ minimizes the number of multi-partition transactions in $\mathcal{W}$; the cost increases from zero as both the number of distributed transactions and the total number of partitions accessed by those transactions increases.

As shown in Algorithm 1, the $CoordinationCost$ function uses the DBMS's internal API function $GetPartitions$ to estimate what partitions each transaction will access [55, 181]. This is the same API that the DBMS uses at run time to determine where to route query requests. For a given design $\mathcal{D}$ and a transaction txn, this function deterministically returns the set of partitions P, where for each $p \in P$ the transaction txn either (1) executed at least one query that accessed p or (2) executed its stored procedure control code at the node managing p (i.e., its base partition). The partitions accessed by txn's queries are calculated by examining the input parameters that reference the tables' partitioning columns in $\mathcal{D}$ (if it is not replicated) in the pre-computed query plans.

There are three cases that $GetPartitions$ must handle for designs that include replicated tables and secondary indexes. First, if a read-only query accesses only replicated tables or indexes, then the query

Algorithm 2 $SkewFactor(\mathcal{D}, \mathcal{W})$

$skew \leftarrow [\,], txnCounts \leftarrow [\,]$
for $i \leftarrow 0$ to $numIntervals$ **do**
  $skew[i] \leftarrow CalculateSkew(\mathcal{D}, \mathcal{W}, i)$
  $txnCounts[i] \leftarrow NumTransactions(\mathcal{W}, i)$
end for
return $\left(\dfrac{\sum_{i=0}^{numIntervals} skew[i] \times txnCounts[i]}{\sum txnCounts} \right)$

executes on the same partition as its transaction's base partition. Next, if a query joins replicated and non-replicated tables, then the replicated tables are ignored and the estimated partitions are the ones needed by the query to access the non-replicated tables. Lastly, if a query modifies a replicated table or secondary index, then that query is broadcast to all of the partitions.

After counting the distributed transactions, the coordination cost is calculated as the ratio of the total number of partitions accessed ($partitionCount$) divided by the total number of partitions that could have been accessed. We then scale this result based on the ratio of distributed to single-partition transactions. This ensures, as an example, that the cost of a design with two transactions that both access three partitions is greater than a design where one transaction is single-partitioned and the other accesses five partitions.

4.4.2 Skew Factor

Although by itself $CoordinationCost$ is able to generate designs that maximize the number of single-partition transactions, it causes the design algorithm to prefer solutions that store the entire database in as few partitions as possible. Thus, we must include an additional factor in the cost model that strives to spread the execution workload uniformly across the cluster.

The function $SkewFactor(\mathcal{D}, \mathcal{W})$ shown in Algorithm 2 calculates how well the design minimizes skew in the database. To ensure that skew measurements are not masked by time, the $SkewFactor$ function divides $\mathcal{W}$ into finite intervals ($numIntervals$) and calculates the final estimate as the arithmetic mean of the skew factors weighted by the number of transactions executed in each interval (to accommodate variable interval sizes). To illustrate why these intervals are needed, consider a design for a two-partition database that causes all of the transactions at time t_1 to execute only on the first partition while the second partition remains idle, and then all of the transactions at time t_2 execute only on the second partition. If the skew is measured as a whole, then the load appears balanced because each partition executed exactly half of the transactions. The value of $numIntervals$ is an administrator-defined parameter. In our evaluation in Section 4.6, we use an interval size that aligns with workload shifts to illustrate that our cost model detects this skew. We leave it as future work to derive this parameter using a pre-processing step that calculates non-uniform windows.

The function $CalculateSkew(\mathcal{D}, \mathcal{W}, interval)$ shown in Algorithm 3 generates the estimated skew factor of $\mathcal{W}$ on $\mathcal{D}$ for the given interval. We first calculate how often partitions are accessed and then determine how much over- or under-utilized each partition is in comparison with the optimal distribution ($best$). To

Algorithm 3 $CalculateSkew(\mathcal{D}, \mathcal{W}, interval)$

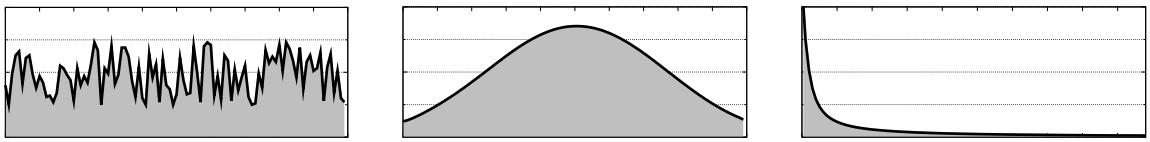
$partitionCounts \leftarrow [\,]$
for all $txn \in \mathcal{W}$, where $txn.interval = interval$ **do**
  for all $p \in GetPartitions(\mathcal{D}, txn)$ **do**
    $partitionCounts\,[p] \leftarrow partitionCounts\,[p] + 1$
  end for
end for
$total \leftarrow \sum partitionCounts$
$best \leftarrow \frac{1}{numPartitions}$
$skew \leftarrow 0$
for $i \leftarrow 0$ to $numPartitions$ **do**
  $ratio \leftarrow \frac{partitionCounts[i]}{total}$
  if $ratio < best$ **then**
    $ratio \leftarrow best + \left(\left(1 - \frac{ratio}{best}\right) \times (1 - best)\right)$
  end if
  $skew \leftarrow skew + \log\left(\frac{ratio}{best}\right)$
end for
return $\left(\dfrac{skew}{\log\left(\frac{1}{best}\right) \times numPartitions}\right)$

ensure that idle partitions are penalized as much as overloaded partitions, we invert any partition estimates that are less than $best$, and then scale them such that the skew value of a ratio as it approaches zero is the same as a ratio as it approaches one. The final normalized result is the sum of all the skew values for each partition divided by the total skew value for the cluster when all but one partition is idle.

Fig. 4.3 shows how the skew factor estimates increase as the amount of skew in the partitions' access distribution increases.

4.4.3 Incomplete Designs

Our cost model must also calculate estimates for designs where not all of the tables and procedures have been assigned an attribute yet [166]. This allows Horticulture to determine whether an incomplete design has a greater cost than the current best design, and thus allows it to skip exploring the remainder of the search tree below its current location. We designate any query that references a table with an unset attribute in a design as being *unknown* (i.e., the set of partitions accessed by that query cannot be estimated). To compute the coordination cost of an incomplete design, we assume that any unknown query is single-partitioned. We take the opposite tack when calculating the skew factor of an incomplete design and assume that all unknown queries execute on all partitions in the cluster. As additional information is added to the design, queries change to a *knowable* state if all of the tables referenced by the query are assigned a partitioning attribute. Any unknown queries that are single-partitioned for an incomplete design $\mathcal{D}$ may become distributed as more variables are bound in a later design $\mathcal{D}'$. But any transaction that is distributed in $\mathcal{D}$ can never become single-partitioned in $\mathcal{D}'$, as this would violate the monotonically increasing cost function requirement of LNS.



(a) Random Skew = 0.34 **(b)** Gaussian Skew = 0.42 **(c)** Zipfian Skew = 0.74

Figure 4.3: Example $CalculateSkew$ estimates for different distributions on the number of times partitions are accessed. Each entry along the x-axis represents a unique partition.

4.5 Optimizations

We now provide an overview of the optimizations that we developed to improve the search time of Horticulture's LNS algorithm. The key to reducing the complexity of finding the optimal database design for an application is to minimize the number of designs that are evaluated [230]. To do this, Horticulture needs to determine which attributes are relevant to the application and are thus good candidates for partitioning. For example, one would not horizontally partition a table by a column that is not used in any query. Horticulture must also discern which relevant attributes are accessed the most often and would therefore have the largest impact on the DBMS's performance. This allows Horticulture to explore solutions using the more frequently accessed attributes first and potentially move closer to the optimal solution more quickly.

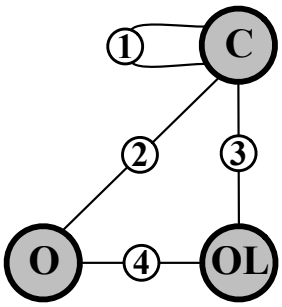
We now describe how to derive such information about an application from its sample workload and store them in a graph structure used in Sections 4.3.1 and 4.3.3. We then present a novel compression scheme for reducing the number of transactions that are examined when computing cost model estimates in Section 4.4.

4.5.1 Access Graphs

Horticulture extracts the key properties of transactions from a workload trace and stores them in undirected, weighted graphs, called *access graphs* [17, 230]. These graphs allow the tool to quickly identify important relationships between tables without repeatedly reprocessing the trace. Each table in the schema is represented by a vertex in the access graph and vertices are adjacent through edges in the graph if the tables they represent are *co-accessed*. Tables are considered co-accessed if they are used together in one or more queries in a transaction, such as in a join. For each pair of co-accessed attributes, the graph contains an edge that is weighted based on the number of times that the queries forming this relationship are executed in the workload trace. A simplified example of an access graph for the TPC-C benchmark is shown in Fig. 4.4. An access graph also contains self-referencing edges for those tables used in single table operations, including all `INSERT`, `UPDATE`, and `DELETE` queries.

We extend prior definitions of access graphs to accommodate stored procedure-based DBMSs. In previous work, an access graph's structure is based on either queries' join relationships [230] or tables' join order in query plans [17]. These approaches are appropriate when examining a workload on a query-by-query basis, but fail to capture relationships between multiple queries in the same transaction, such as a logical join operation split into two or more queries—we call this an *implicit reference*.

To discover these implicit references, Horticulture uses a workload's parameter mappings [181] to determine whether a transaction uses the same input parameters in multiple query invocations. Since implicit



Edge#	Columns	Weight
(1)	C.C_ID ↔ C.C_ID	200
(2)	C.C_ID ↔ O.O_C_ID	100
(3)	O.O_ID ↔ OL.OL_O_ID	100
(4)	O.O_ID ↔ OL.OL_O_ID O.O_C_ID ↔ OL.OL_C_ID	100

Figure 4.4: An access graph derived from a workload trace.

reference edges are derived from multiple queries, their weights are based on the minimum number of times those queries are all executed in a single transaction [230].

A simplified example of an access graph for the TPC-C benchmark is shown in Fig. 4.4. To create this graph from a workload trace, we first insert a new vertex v_i in the graph for every unique table $\mathcal{T}_i$ that is referenced in all of the stored procedures' queries. Two vertices v_i, v_j are adjacent through an edge $e_{i,j}$ if the tables $\mathcal{T}_i$ and $\mathcal{T}_j$ are co-accessed either explicitly (i.e., using a single query that joins them) or implicitly (i.e., using two separate queries that each only access one of the tables but use input parameters that are mapped together). Fig. 4.4 shows that there is an edge (#3) incident to CUSTOMER and ORDERS, meaning that the trace contains queries that co-access the columns C_ID and O_C_ID. Access graphs can be built in a single pass over the trace and allows Horticulture to scale easily with larger trace sizes.

4.5.2 Workload Compression

Using large sample workloads when evaluating a potential design improves the cost model's ability to estimate the target database's properties. But the cost model's computation time depends on the sample workload's size (i.e., the number of transactions) and complexity (i.e., the number of queries per transaction). Existing design tools employ random sampling to reduce workload size [66], but this approach can produce poor designs if the sampling masks skew or other potentially valuable information about the workload [57]. We instead use an alternative approach that compresses redundant transactions and redundant queries without sacrificing accuracy. Our scheme is more efficient than previous methods in that we only consider what tables and partitions that queries access, rather than the more expensive task of comparing sets of columns [57, 79].

Compressing a transactional workload is a two-step process. First, we combine sets of similar queries in individual transactions into fewer weighted records [79]. Such queries often occur in stored procedures that contain loops in their control code. After combining queries, we then combine similar transactions into a smaller number of weighted records in the same manner. The cost model will scale its estimates using these weights without having to process each of the records separately in the original workload.

To identify which queries in a single transaction are combinable, we compute the *input signature* for each query from the values of its input parameters and compare it with the signature of all other queries. A query's input signature is an unordered list of pairs of tables and partition ids that the query would access if each table is horizontally partitioned on a particular column. As an example, consider the following query on the

CUSTOMER table:

```
SELECT * FROM CUSTOMER WHERE C_ID = 10 AND C_LAST = "Smith"
```

Assuming that the input value "10" corresponds to partition #10 if the table was partitioned on C_ID and the input value "Smith" corresponds to partition #3 if it was partitioned on C_LAST, then this query's signature is $\{(\mathtt{C}, 10), (\mathtt{C}, 3)\}$. We only use the parameters that are used with co-accessed columns when computing the signature. For example, if only C_ID is referenced in the access graph, then the above example's input signature is $\{(\mathtt{C}, 10)\}$.

Each transaction's input signature includes the query signatures computed in the previous step, as well as the signature for the transaction's procedure input parameters. We use the parameter mappings [181] to find which of the tables' columns are used in predicates that are linked to procedure input parameters, and then prune any columns that are not referenced in the access graph. Any set of transactions with the same query signatures and procedure input parameter signature are combined into a single weighted record.

4.6 Experimental Evaluation

To evaluate the effectiveness Horticulture's design algorithms, we integrated our tool with H-Store and ran several experiments that compare our approach to alternative approaches. These other algorithms include a state-of-the-art academic approach, as well as other solutions commonly applied in practice:

HR+: Our large-neighborhood search algorithm from Section 4.3.

HR–: Horticulture's baseline iterative greedy algorithm, where design options are chosen one-by-one independently of others.

SCH: The Schism [66] graph partitioning algorithm.

PKY: A simple heuristic that horizontally partitions each table based on their primary key.

MFA: The initial design algorithm from Section 4.3.1 where options are chosen based on how frequently attributes are accessed.

4.6.1 Benchmark Workloads

We now describe the workloads from H-Store's built-in benchmark framework that we used in our evaluation. See Appendix A for a full description of each workload. The size of each database in these experiments is approximately 1GB per partition.

TATP: This is an OLTP testing application that simulates a typical caller location system used by telecommunication providers [226]. It consists of four tables, three of which are foreign key descendants of the root SUBSCRIBER table. Most of the stored procedures in TATP have a SUBSCRIBER id as one of their input parameters, allowing them to be routed directly to the correct node.

TPC-C: This is the current industry standard for evaluating the performance of OLTP systems [216]. It consists of nine tables and five stored procedures that simulate a warehouse-centric order processing application. All of the procedures in TPC-C provide a warehouse id as an input parameter for the transaction, which is the foreign key ancestor for all tables except ITEM.

TPC-C (Skewed): Our benchmarking infrastructure also allows us to tune the access skew for benchmarks. In particular, we generated a temporally skew load for TPC-C, where the `WAREHOUSE id` used in the transactions' input parameters is chosen so that at each time interval all of the transactions target a single warehouse. This workload is uniform when observed globally, but at any point in time there is a significant amount of skew. This help us to stress-test our system when dealing with temporal-skew, and to show the potential impact of skew on the overall system throughput.

SEATS: This benchmark models an on-line airline ticketing system where customers search for flights and make reservations [105]. It consists of eight tables and six stored procedures. The benchmark is designed to emulate a back-end system that processes requests from multiple applications that each provides disparate inputs. Thus, many of its transactions must use secondary indexes or joins to find the primary key of a customer's reservation information. For example, customers may access the system using either their frequent flyer number or customer account number. The non-uniform distribution of flights between airports also creates imbalance if the database is partitioned by airport-derived columns.

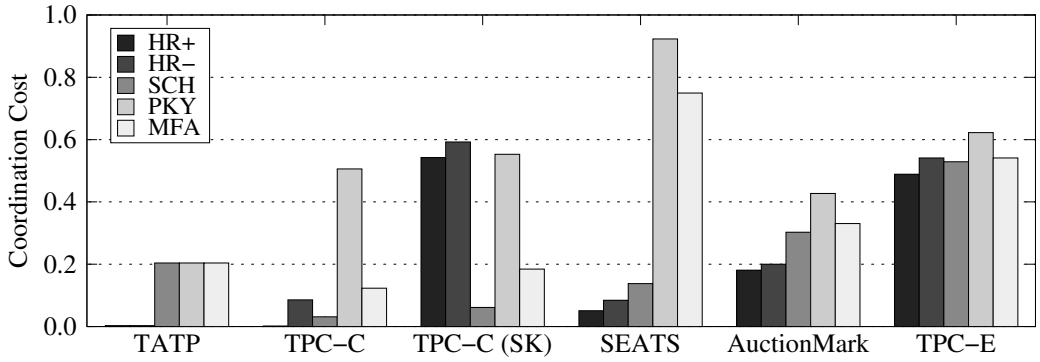
AuctionMark: This is a 16-table benchmark based on an Internet auction system [104]. Most of its 10 procedures involve an interaction between a buyer and a seller. The user-to-item ratio follows a Zipfian distribution, which means that there are a small number of users that are selling a large portion of the total items. The total number of transactions that target each item is temporally skewed, as items receive more activity (i.e., bids) as the auction approaches its closing time. It is difficult to generate a design for AuctionMark that includes stored procedure routing because several of the benchmark's procedures include conditional branches that execute different queries based on the transaction's input parameters.

TPC-E: Lastly, the TPC-E benchmark is the successor of TPC-C and is designed to reflect the workloads of modern OLTP applications [215]. Its workload features 12 stored procedures, 10 of which are executed in the regular transactional mix while two are periodically executed "clean-up" procedures. Unlike the other benchmarks, many of TPC-E's 33 tables have foreign key dependencies with multiple tables, which create conflicting partitioning candidates. Some of the procedures also have optional input parameters that cause transactions to execute mutually exclusive sets of queries based on which of these parameters are given at run time.

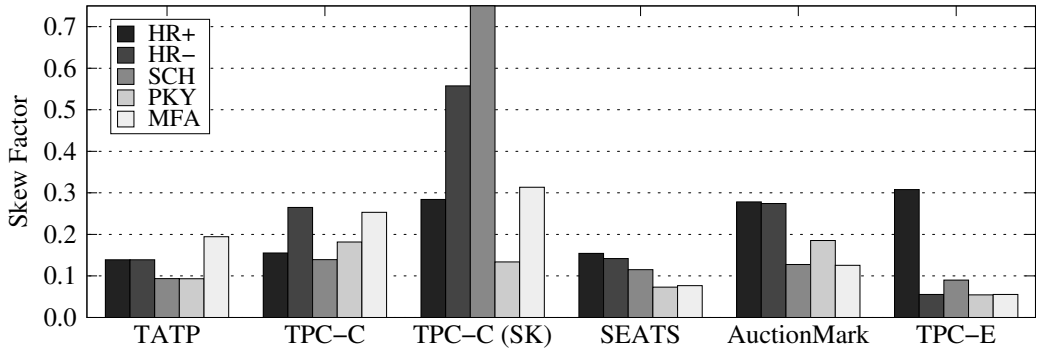
4.6.2 Design Algorithm Comparison

The first experiment that we present is an off-line comparison of the database design algorithms listed above. We execute each algorithm for all of the benchmarks to generate designs for clusters ranging from four to 64 partitions. Each algorithm is given an input workload trace of 25k transactions, and then is tested using a separate trace of 25k transactions. We evaluate the effectiveness of the designs of each algorithm by measuring the number of distributed transactions and amount of skew in those designs over the test set.

Fig. 4.5a shows that HR+ produces designs with the lowest coordination cost for every benchmark except TPC-C (Skewed), with HR– and SCH designs only slightly higher. Because fewer partitions are accessed using HR+'s designs, the skew estimates in Fig. 4.5b greater (this why the cost model uses the α and β



(a) The estimated coordination cost for the benchmarks.



(b) The estimated skew of the transactions' access patterns.

Figure 4.5: Off-line measurements of the designs algorithms in Section 4.6.2.

parameters). We ascribe the improvements of HR+ over HR− and MFA to the LNS algorithm's effective exploration of the search space using our cost model and escaping local minima.

For TPC-C (Skewed), HR+ chooses a design that increases the number of distributed transactions in exchange for a more balanced load. Although the SCH algorithm does accommodate skew when selecting a design, it currently does not support the temporal skew used in this benchmark. The skew estimates for PKY and MFA are lower than others in Fig. 4.5b because more of the transactions touch all of the partitions, which causes the load to be more uniform.

4.6.3 Transaction Throughput

The next experiment is an end-to-end test of the quality of the designs generated in the previous experiment. We compare the designs from our best algorithm (HR+) against the state-of-the-art academic approach (SCH) and the best baseline practical solution (MFA). We execute select benchmarks in H-Store using the designs for these algorithms and measure the system's overall throughput.

We execute each benchmark using five different cluster sizes of Amazon EC2 nodes allocated within a single region. Each node has eight virtual cores and 70GB of RAM (`m2.4xlarge`). We assign at most seven partitions per node, with the remaining partition reserved for the networking and administrative functionalities of H-Store. The execution engine threads are given exclusive access to a single core to improve cache locality.

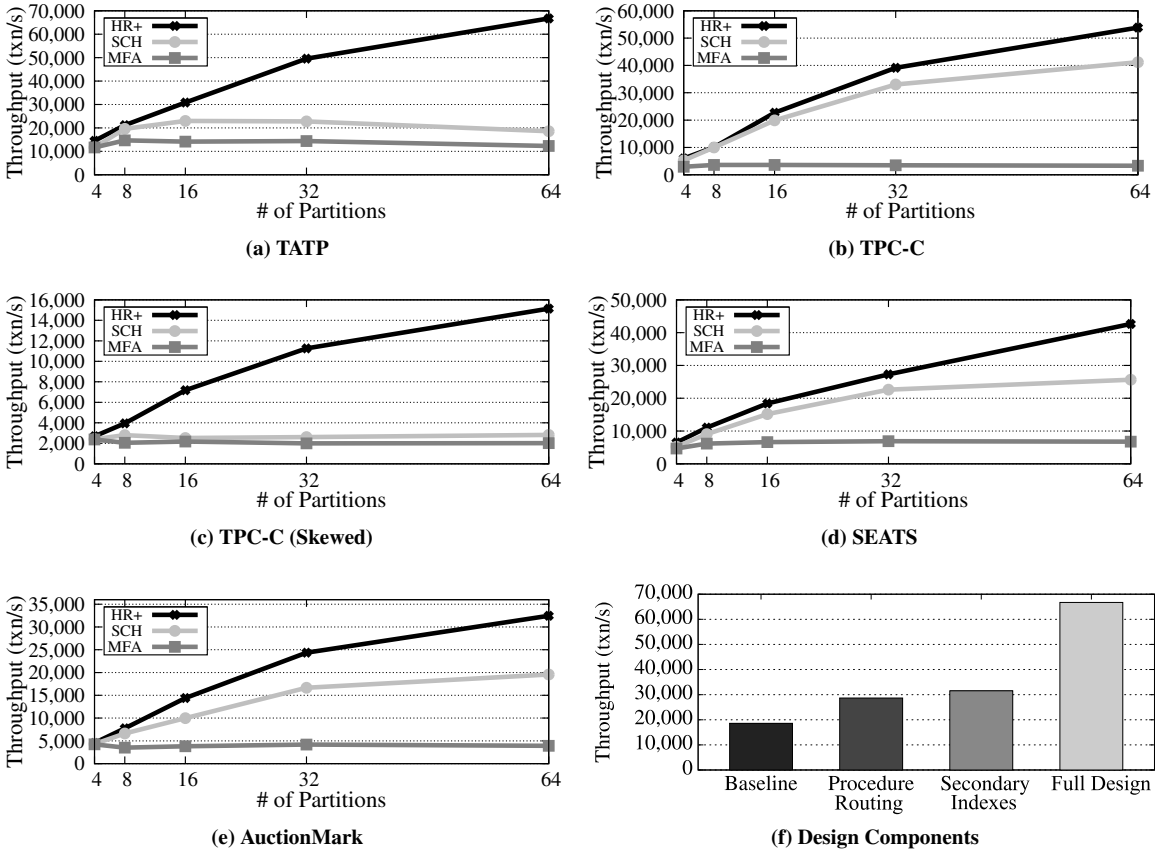


Figure 4.6: Transaction throughput measurements for the HR+, SCH, and MFA design algorithms.

Transaction requests are submitted from up to 5000 simulated client terminals running on separate nodes in the same cluster. Each client submits transactions to any node in the H-Store cluster in a closed loop: after it submits a request, it blocks until the result is returned. Using a large number of clients ensures that the execution engines' workload queues are never empty.

We execute each benchmark three times per cluster size and report the average throughput of these trials. In each trial, the DBMS "warms-up" for 60 seconds and then the throughput is measured after five minutes. The final throughput is the number of transactions completed in a trial run divided by the total time (excluding the warm-up period). H-Store's benchmark framework ensures that each run has the proper distribution of executed procedures according to the benchmark's specification.

All new requests are executed in H-Store as single-partitioned transactions with reduced concurrency control protection; if a transaction attempts to execute a multi-partition query, then it is aborted and restarted with full concurrency control. Since SCH does not support stored procedure routing, the system is unable to determine where to execute each transaction request even if the algorithm generates the optimal partitioning scheme for tables. Thus, to obtain a fair comparison of the two approaches, we implemented a technique from IBM DB2 [59] in H-Store to handle this scenario. Each transaction request is routed to a random node

by the client where it will start executing. If the first query that the transaction dispatches attempts to access data not stored at that node, then it is aborted and re-started at the proper node. This ensures that single-partition transactions execute with reduced concurrency control protection, which is necessary for achieving good throughput in H-Store.

The throughput measurements in Fig. 4.6 show that the designs generated by HR+ improve the throughput of H-Store by factors $1.3\times$ to $4.3\times$ over SCH and $1.1\times$ to $16.3\times$ over MFA. This validates two important hypotheses: (1) that our cost model and search technique are capable of finding good designs, and (2) that by explicitly accounting for stored procedure routing, secondary indexes replication, and temporal-skew management, we can significantly improve over previous best-in-class solutions. Other notable observations are that (1) the results for AuctionMark highlight the importance of stored procedure routing, since this is the only difference between SCH and HR+, (2) the TATP, SEATS, and TPC-C experiments demonstrate the combined advantage of stored procedures and replicated secondary indexes, and (3) that TPC-C (Skewed) illustrates the importance of mitigating temporal-skew. We also note that the performance of H-Store is less than expected for larger cluster sizes due to clock skew issues when choosing transaction identifiers that ensure global ordering [211].

For this last item, we note that TPC-C (Skewed) is designed to stress-test the designs algorithms under extreme temporal-skew conditions to evaluate its impact on throughput; we do not claim this to be a common scenario. In this setting, any system ignoring temporal-skew will choose the same design used in Fig. 4.6b, resulting in near-zero scale-out. Fig. 4.6b shows that both SCH and MFA do not improve performance as more nodes are added to the cluster. On the contrary, HR+ chooses a different design (i.e., partitioning by `WAREHOUSE id` and `DISTRICT id`), thus accepting many more distributed transactions in order to reduce skew. Although all the approaches are affected by skew resulting in an overall lower throughput, HR+ is significantly better with more than $6\times$ throughput increase for the same $8\times$ increase in nodes.

To further ascertain the impact of the individual design elements, we executed TATP again using the HR+ design but alternatively removing: (1) client-side stored procedure routing (falling back on the redirection mechanism we built to test SCH), (2) the secondary indexes replication, or (3) both. Fig. 4.6f shows the relative contributions with stored procedure routing delivering 54.1% over the baseline approach (that otherwise coincide with the one found by SCH), secondary indexes contribute 69.6%, and combined they deliver a $3.5\times$ improvement. This is because there is less contention for locking partitions in the DBMS's transaction coordinators [131].

4.6.4 Cost Model Validation

Horticulture's cost model is not meant to provide exact throughput predictions, but rather to quickly estimate the relative ordering of multiple designs. To validate that these estimates are correct, we tested its accuracy for each benchmark and number of partitions by comparing the results from Fig. 4.5 and Fig. 4.6. We note that our cost model predicts which design is going to perform best in 95% of the experiments. In the cases where the cost model fails to predict the optimal design, our analysis indicates that they are inconsequential because they are from workloads where the throughput results are almost identical (e.g., TATP on four partitions). We suspect that the throughput differences might be due to transitory EC2 load conditions rather than actual difference in the designs. Furthermore, the small absolute difference indicates that such errors will not

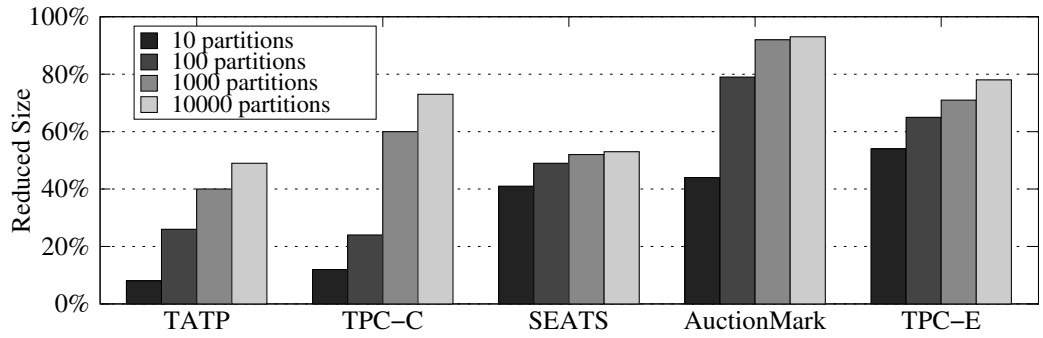


Figure 4.7: Workload Compression Rates

significantly degrade performance.

4.6.5 Compression & Scalability

We next measured the workload compression rate for the scheme described Section 4.5.2 using the benchmark's sample workloads when the number of partitions increases exponentially. The results in Fig. 4.7 show that the compression rate decreases for all of the benchmarks as the number of partitions increases due to the decreased likelihood of duplicate parameter signatures. The workload for the TPC-C benchmark does not compress well due to greater variability in the procedure input parameter values.

We also analyzed Horticulture's ability to generate designs for large cluster sizes. The results in Fig. 4.8 shows that the search time for our tool remains linear as the size of the database increases.

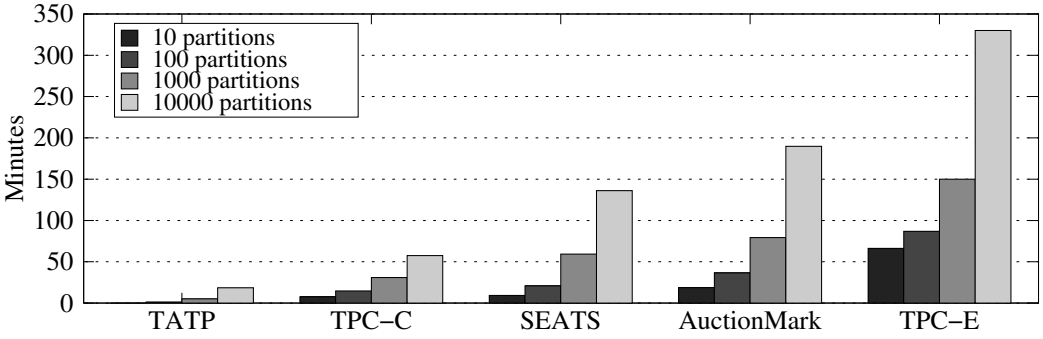
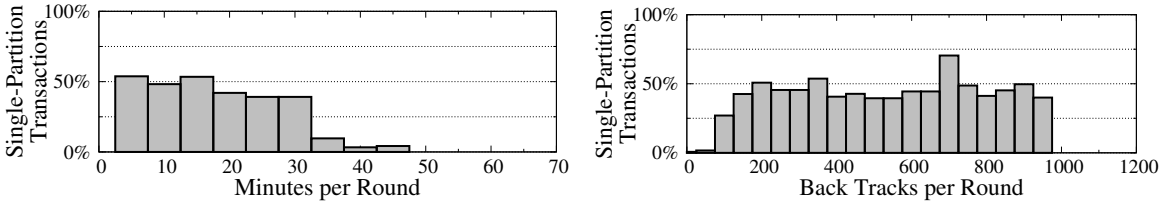


Figure 4.8: LNS search time for different cluster sizes

4.6.6 Search Parameter Sensitivity Analysis

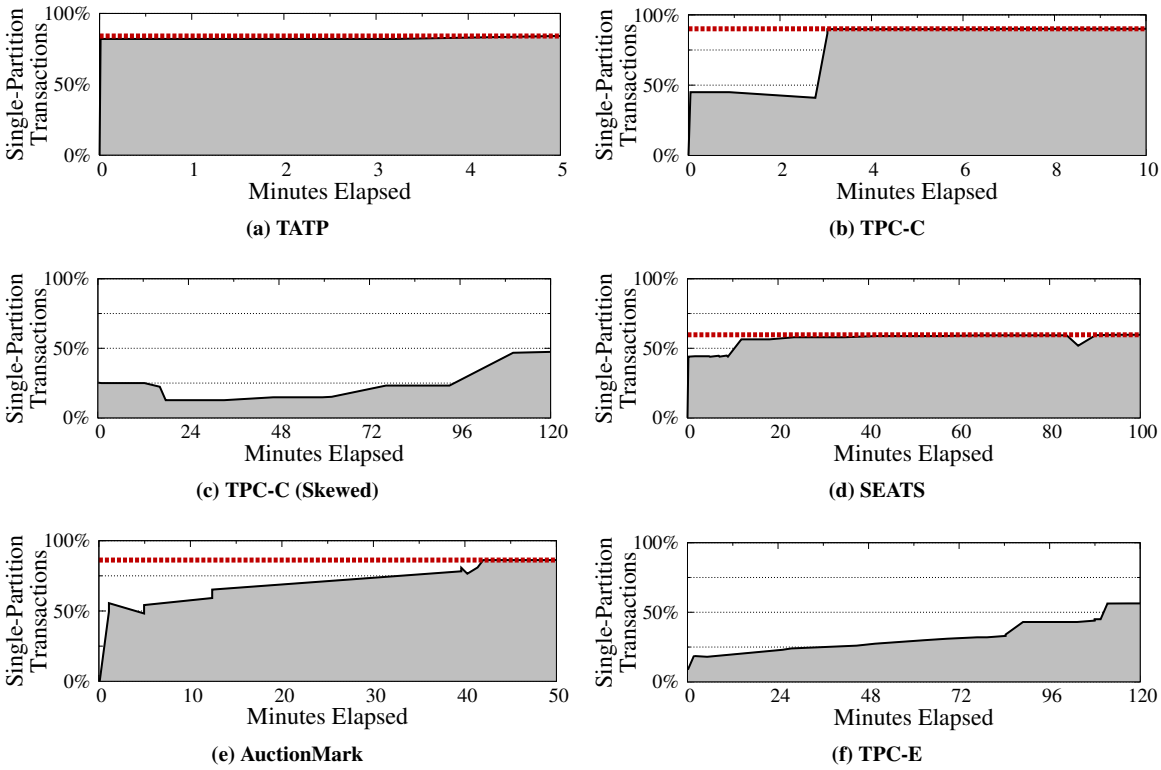
As discussed in Section 4.3.3, there are parameters that control the run time behavior of Horticulture: each local search round executes until either it (1) exhausts its time limit or (2) reaches its backtrack limit. Although Horticulture dynamically adjusts these parameters [86], their initial values can affect the quality of the designs found. For example, if the time limit is too small, then Horticulture will fail to fully explore each neighborhood. Moreover, if it is too large, then too much time will be spent exploring neighborhoods that never yield a better design. The LNS algorithm will continue looking for a better design until either it (1) surpasses the total amount of time allocated by the administrator or (2) has exhausted the search space. In



(a) Local Search Times

(b) Backtrack Limits

Figure 4.9: A comparison of LNS-generated designs for TPC-E using different (a) local search times and (b) backtrack limits.



(a) TATP

(b) TPC-C

(c) TPC-C (Skewed)

(d) SEATS

(e) AuctionMark

(f) TPC-E

Figure 4.10: The best solution found by Horticulture over time. The red dotted lines represent known optimal designs (when available).

this experiment, we investigate what are good default values for these search parameters.

We first experimented with using different local search and backtrack limits for the TPC-E benchmark. We chose TPC-E because it has the most complex schema and workload. We executed the LNS algorithm for two hours using different local search time limits with an infinite backtrack limit. We then repeated this experiment using an infinite local search time limit but varying the backtrack limit. The results in Fig. 4.9 show that using the initial limits of approximately five minutes and 100–120 backtracks produces designs with lower costs more quickly.

Non-deterministic algorithms, such as LNS, are not guaranteed to discover the optimal solution, which

means that there is no way for the administrator to know how much time to let Horticulture to continue searching. Thus, we need a way to know when to stop searching for a better design. A naïve approach is to halt when the algorithm fails to find a new solution after a certain amount of time. But this is difficult to estimate for arbitrary inputs, since the search time is dependent on a number of factors.

Another approach is to calculate a lower bound using a theoretical design [166] and then halt the LNS algorithm when it finds a design with a cost that is within a certain distance to that bound [86]. We compute this bound by estimating the cost of the workload using a design where all transactions execute as single-partitioned and with no skew in the cluster (i.e., round-robin assignment). Note that such a design is likely infeasible, since partitioning a database to make every transaction single-partitioned cannot always be done without making other transactions distributed. The graphs in Fig. 4.10 show the amount of time it takes for the LNS algorithm to find solutions that converge towards the lower bound. We show the quality of the design in terms of single-partition transactions as time proceeds. The red dotted line in each graph represents the best known design we are aware of for each benchmark—we do not have reference designs for TPC-C (Skewed) and TPC-E, other than the one found by our tools. The cost improvements shown in the graphs plateau after a certain point, which is the desired outcome.

Overall, these experiments show that to achieve great performance for enterprise-class OLTP workloads, especially on modern NewSQL systems, it is paramount that a design tool supports stored procedures, replicated secondary indexes, and temporal skew. To the best of our knowledge, Horticulture is the first to consider all of these issues.

4.7 Conclusion

We presented a new approach for automatically partitioning a database in a shared-nothing, distributed DBMS. Our algorithm uses a large-neighborhood search technique together with an analytical cost model to minimize the number of distributed transactions while controlling the amount of skew. To the best of our knowledge, the system that we present is the first to target enterprise OLTP systems by supporting stored procedure routing, replicated secondary indexes, and temporal-skew handling. We experimentally prove that these options are important in distributed OLTP systems, and that our approach generates database designs that enable improve performance by up to $16\times$ over other solutions.

Chapter 5

Predictive Transaction Modeling

Even after generating a database design using the techniques described in Chapter 4, achieving good performance in distributed DBMSs still requires significant tuning because of distributed transactions that access multiple partitions. Such transactions require the DBMS to either (1) block other transactions from using each partition until that transaction finishes or (2) use fine-grained locking with deadlock detection to execute transactions concurrently [131]. In either strategy, the DBMS may also need to maintain an undo buffer in case the transaction aborts. Avoiding such onerous concurrency control is important (cf. Section 2.2) [112]. To do so, however, requires the DBMS to have additional information about transactions before they start. For example, if the DBMS knows that a transaction only needs to access data at one partition, then that transaction can be redirected to the machine with that data and executed without heavy-weight concurrency control schemes [211].

It is not practical, however, to require users to explicitly inform the DBMS how individual transactions are going to behave. This is especially true for complex applications where a change in the database's configuration, such as its partitioning scheme, affects transactions' execution properties. Hence, we now present a novel method to automatically select which optimizations the DBMS can apply to transactions at runtime using Markov models. A Markov model is a probabilistic model that, given the current state of a transaction (e.g., which query it just executed), captures the probability distribution of what actions that transaction will perform in the future. Based on this prediction, the DBMS can then enable the proper optimizations. Our approach has minimal overhead, and thus it can be used on-line to observe requests to make immediate predictions on transaction behavior without additional information from the user. We assume that the benefit outweighs the cost when the prediction is wrong. This work is focused on stored procedure-based transactions, which have four properties that can be exploited if they are known in advance: (1) how much data is accessed on each node, (2) what partitions will the transaction read/write, (3) whether the transaction could abort, and (4) when the transaction will be finished with a partition.

In describing this work, we begin with an overview of the optimizations used to improve the throughput of OLTP workloads. We then describe our primary contribution: representing transactions as Markov models in a way that allows a DBMS to decide which of these optimizations to employ based on the most likely behavior of a transaction. Next, we present *Houdini*, an on-line framework that uses these models to generate

predictions about transactions before they start. We have integrated this framework into H-Store and measure its ability to optimize three OLTP benchmarks. The results from these experiments demonstrate that our models select the proper optimizations for 93% of transactions and improve the throughput of the system by 41% on average with an overhead of 5% of the total transaction execution time.

5.1 Runtime Transaction Optimizations

We first discuss the optimizations that are possible if one knows what a transaction will do prior to its execution in a stored procedure-based DBMS. Stored procedures are an effective way to optimize OLTP applications because they reduce the number of round-trips between the client and the database, thereby eliminating most network overhead and shrinking the window for lock contention. They contain parameterized queries separated by *control code* (i.e., application logic), and thus most DBMSs do not know what each transaction invocation of a procedure will do at run time (e.g., what set of pre-defined queries it will execute and what partitions those queries will access). This is because the procedure can contain loops and conditionals that depend on the parameters from the application and the current values stored in the database.

We now discuss the four transaction optimizations that OLTP systems like H-Store can employ at run time if they know certain properties about transactions before they begin to execute.

OP1. Execute the transaction at the node with the partition that it will access the most.

When a new transaction request is received, the DBMS's transaction coordinator must determine which node in the cluster should execute the procedure's control code and dispatch queries. In most systems, this node also manages a partition of data. We call this the *base partition* for a transaction. The "best" base partition is the one containing most of the data that will be accessed by that transaction, as that reduces the amount of data movement. Any transaction that needs to access only one data partition is known as a *single-partition* transaction. These transactions can be executed efficiently on a distributed DBMS, as they do not require multi-node coordination [211]. Hence, determining the correct base partition will dramatically increase throughput and decrease latency in any distributed database that supports stored procedures.

One naïve strategy is to execute each transaction on a random partition to evenly distribute work, but the likelihood that this approach picks the "wrong" partition increases with the number of partitions. An alternative approach, used by IBM's DB2, is to execute the procedure on any node, then if the first statement accesses data in some other partition, abort and re-start the transaction there [59]. This heuristic does not work well, however, for transactions where the first statement accesses data in the wrong partition or a large number of partitions all at once.

OP2. Lock only the partitions that the transaction accesses.

Similarly, knowing all of the partitions that each transaction will access allows the DBMS to avoid traditional concurrency control. If a single-partition transaction will only access data from its base partition, then it can be executed to completion without any concurrency control. Otherwise, the DBMS will "lock" the minimum partitions needed before the transaction starts; partitions that are not involved will process other transactions. Accurately predicting which partitions are needed allows the DBMS to avoid the overhead of

deadlock detection and fine-grained row-based locking [131]. But if a transaction accesses an extra partition that was not predicted, then it must be aborted and re-executed. On the other hand, if the DBMS predicts that a transaction will access multiple partitions but only ends up accessing one, then resources are wasted by keeping unused partitions locked.

OP3. Disable undo logging for non-aborting transactions.

Since a distributed DBMS replicates state over multiple nodes, persistent logging in these environments is unnecessary [19, 112]. These systems instead employ a transient undo log that is discarded once the transaction has committed [211]. The cost of maintaining this log per transaction is large relative to its overall execution time, especially for those transactions that are unlikely to abort (excluding DBMS failures). Thus, if the DBMS can be guaranteed that a transaction will never abort after performing a write operation, then logging can be disabled for that transaction. This optimization must be carefully enabled, however, since the node must halt if a transaction aborts without undo logging.

This optimization is applicable to all main-memory DBMSs, as undo logging is only needed to abort a transaction and not for recovery as used in disk-based systems. This also assumes that that each procedure's control code is robust and will not abort due to programmer error (e.g., divide by zero).

OP4. Speculatively commit the transaction at partitions that it no longer needs to access.

The final optimization that we consider is using speculative execution when a distributed transaction is finished at a partition. For distributed transactions, many DBMSs use two-phase commit to ensure consistency and atomicity. This requires an extra round of network communication: the DBMS sends a prepare message to all partitions and must wait for all of the acknowledgements before it can inform the client that the transaction committed. If the DBMS can identify that a particular query is the last operation that a transaction will perform at a partition, then that query and the prepare message can be combined. This is called the "early prepare" or "unsolicited vote" optimization, and has been shown to improve both latency and throughput in distributed systems [193].

Once a node receives this early prepare for the distributed transaction, the DBMS can begin to process other queued transactions at that node [37, 131]. If these speculatively executed transactions only access tables not modified by the distributed transaction, then they will commit immediately once they are finished. Otherwise, they must wait until the distributed transaction commits. This optimization is similar to releasing locks early in traditional databases' two-phase commit prepare phase [72].

Predicting whether a query is the last one for a given partition is not straightforward for the traditional "conversational" interface because the DBMS does not know what the clients will send next. But even for stored procedures this is not easy, as conditional statements and loops make it non-trivial to determine which queries will be executed by the transaction. As with the other optimizations, the DBMS will have to undo work if it is wrong. If a transaction accesses a partition that it previously declared to be finished with, then that transaction and all speculatively executed transactions at the partition are aborted and restarted.

To demonstrate how the above optimizations improve transaction throughput, we consider an example

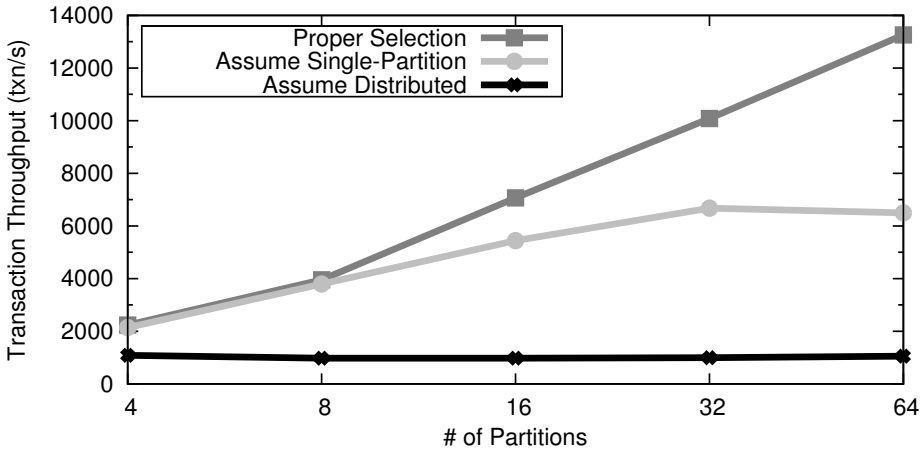


Figure 5.1: The throughput of the system on different partition sizes using three different execution scenarios: (1) All transactions are executed as distributed; (2) All transactions are executed as single-partitioned, distributed transactions are restarted; (3) Single-partition transactions run without concurrency control and distributed transactions lock the minimum number of partitions.

from the TPC-C benchmark [216]. A simplified version of the TPC-C `NewOrder` stored procedure is shown in Fig. 5.2. Approximately 90% of the `NewOrder` requests create orders using items from a single warehouse. If the database is partitioned by warehouse ids (`w_id`), then most of these requests are executed as single-partitioned transactions [211].

We executed `NewOrder` transactions using H-Store in three different ways: (1) all requests are assumed to be distributed and are executed on a random node locking all partitions; (2) all requests are assumed to be single-partitioned and are executed on a random node, and if the transaction tries to access multiple partitions it is aborted and restarted as a distributed transaction that locks the partitions it tried to access before it was aborted; and (3) the client provides the system with the partitions needed for each request and whether it will abort, and the DBMS only locks the necessary partitions. This last case is the best possible scenario for the DBMS. We execute each configuration using five different cluster sizes, with two partitions/warehouses assigned per node. Transaction requests are submitted from clients executing on separate machines in the cluster. Each trial is executed three times and we report the average throughput of the three runs. Note that this experiment only utilizes optimizations **OP1** and **OP2**; we discuss all of the optimizations together in Section 5.5.

The results in Fig. 5.1 show the significance of knowing what a transaction will do before it executes in a system like H-Store. The throughput for the "assume distributed" case is constant for all cluster sizes because the DBMS is limited to the rate that it can send and receive the two-phase commit acknowledgements. When there are only a small number of partitions, the other strategies are roughly equivalent because the likelihood that a transaction is on the partition that has the data it needs is higher. The throughput of H-Store, however, scales better when the system has the proper information before a transaction begins, as opposed to restarting a transaction once it deviates from the single-partitioned assumption.

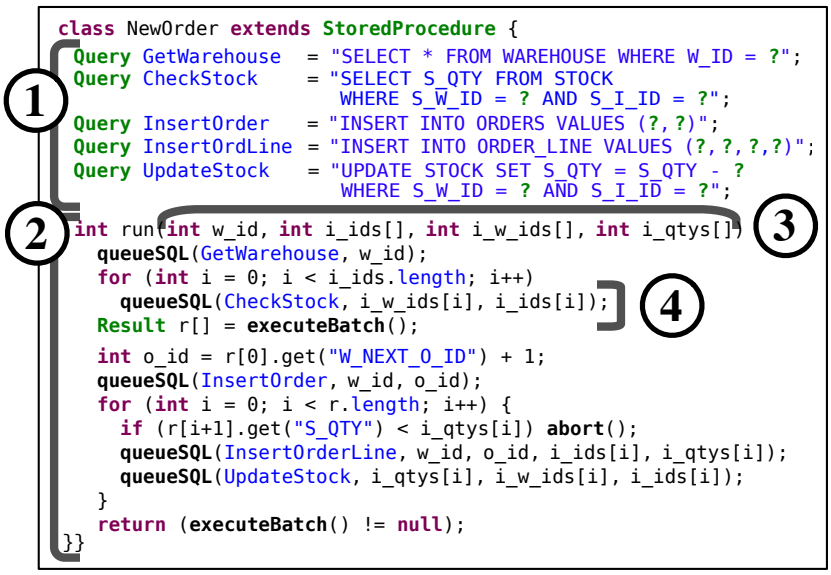


Figure 5.2: A stored procedure defines (1) a set of parameterized queries and (2) control code. For each new transaction request, the DBMS invokes the procedure's `run` method and passes in (3) the procedure input parameters sent by the client. The transaction invokes queries by passing their unique handle to the DBMS along with the values of its (4) query input parameters.

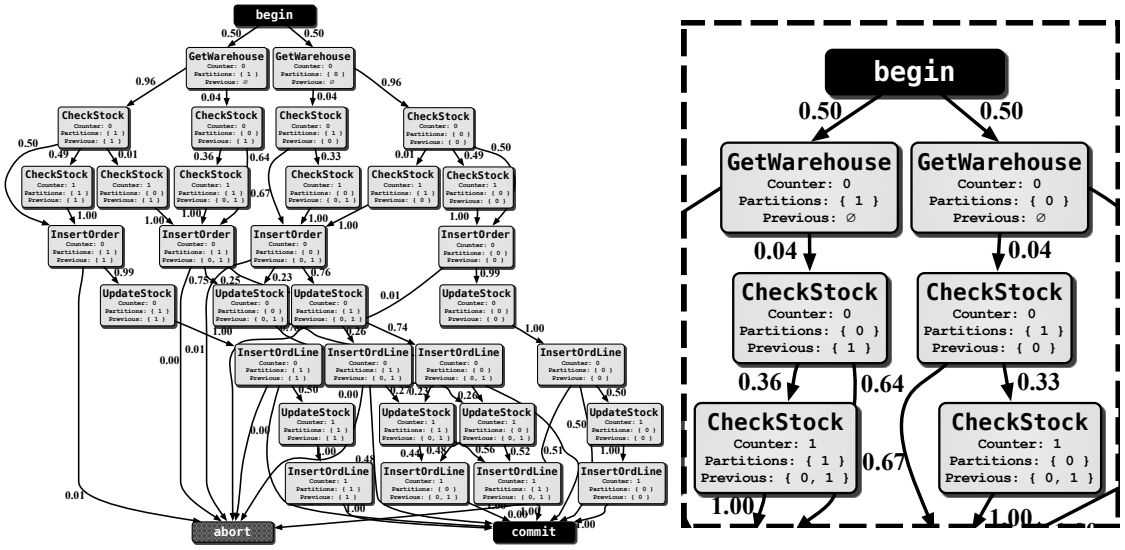
5.2 Transaction Models

The throughput improvements in the previous experiment require the application to specify exactly which partitions will be accessed and whether the transaction will abort, which depends on how the data is partitioned and the state of the database. This adds additional burden on developers. Worse, this will change any time the database is reorganized. An alternative approach is to model transactions in such a way that allows the DBMS automatically extract properties for each new transaction and then dynamically enable optimizations without needing to modify the application's code.

Markov models are an excellent fit for our problem because they can be both generated quickly and used to estimate transaction properties without expensive computations [121]. The latter is important for OLTP systems, since it is not useful to spend 50 ms deciding which optimizations to enable for a 10 ms transaction. In this section, we define our transaction Markov models and outline how they are generated. We describe how to use these models to select optimizations, as well as how to maintain them, in subsequent sections.

5.2.1 Definition

Stored procedures are composed of a set of queries that have unique names. A given invocation of a stored procedure executes a subset of these queries in some order, possibly repeating queries any number of times due to loops. For a stored procedure $\mathcal{SP}_\ell$, we define the transaction Markov model $\mathcal{M}_\ell$ as an acyclic directed graph of the execution states and paths of $\mathcal{SP}_\ell$. An *execution state* is defined as a vertex $v_i \in V(\mathcal{M}_\ell)$ that represents a unique invocation of a single query within $\mathcal{SP}_\ell$, where v_i is identified by (1) the name of the query, (2) the number of times that the query has been executed previously in the transaction (`counter`), (3)



(a) Markov Model **(b)** Detailed View

Figure 5.3: An example of a Markov model for the `NewOrder` stored procedure shown in Fig. 5.2. The full model with all of the possible execution states is shown in Fig. 5.3a. Fig. 5.3b shows a detailed view of the boxed region in the larger graph.

the set of partitions that this query will access (`partitions`) as returned by the DBMS's internal API [55], and (4) the set of partitions that the transaction has already accessed (`previous`). In essence, a vertex encodes all of the relevant execution history for a transaction up to that point. Each model also contains three vertices that represent the `begin`, `commit`, and `abort` states of a transaction. The two vertices v_i, v_j are adjacent in $\mathcal{M}_\ell$ through the directed edge $e_{i,j} \in E(\mathcal{M}_\ell)$ if a transaction executes v_j's query immediately after executing v_i's query.

The outgoing edges from a vertex $v_i \in V(\mathcal{M}_\ell)$ represent the probability distribution that a transaction transitions from v_i's state to one of its subsequent states. If a transaction committed, then the vertex for the last query it executed is connected by an edge to the `commit` state; in the same way, if the transaction aborted, then the last query's vertex is connected to the `abort` state. A transaction's *execution path* in $\mathcal{M}_\ell$ is an ordered list of vertices from the `begin` state to one of these two terminal states.

These Markov models are used to predict the future states of new transactions based on the history of previous transactions. Each model is generated from a sample *workload trace* for an application. A trace contains for each transaction (1) its procedure input parameters and (2) the queries it executed, with their corresponding parameters. Because the trace does not encode what partitions each query accessed, new models must be regenerated from the trace whenever the database's partitioning scheme changes.

Fig. 5.3 shows an example of a Markov model for the `NewOrder` procedure in Fig. 5.2. In the detailed view shown in Fig. 5.3b, we see that there are two `GetWarehouse` vertices that are adjacent to the `begin` vertex. The sets of previously accessed partitions for these vertices are empty since they are the first query in the transaction, while the subsequent `CheckStock` states include partitions that were touched by their parent vertices. For simplicity, Fig. 5.3 was generated for a database that has only two partitions, and thus every

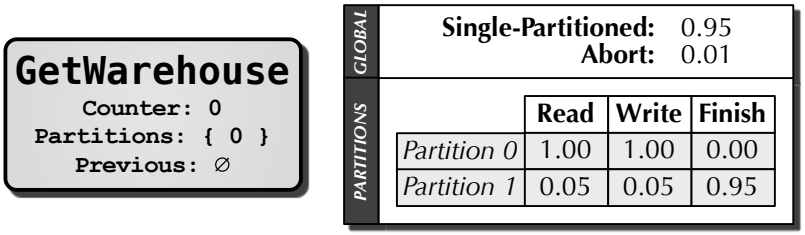


Figure 5.4: The probability table for the `GetWarehouse` state from Fig. 5.3. The table shows that with 100% certainty any transaction that reaches this state will execute another query that accesses partition #0 before it commits. Conversely, there is a 5% chance that it will need to either read or write data on partition #1.

`NewOrder` transaction executes the `GetWarehouse` query on just one of the two partitions (assuming that each warehouse is assigned to one partition). The `begin` state's edge probabilities are equal, which means that transactions will execute the `GetWarehouse` query on either partition with equal probability.

These edge weights are used to calculate the confidence level of an execution path and predict what a transaction will do when its procedure input parameters is insufficient to predict what query a transaction will execute next (see Section 5.3.2).

Every vertex is also annotated with a table of probabilities for events that may occur after the transaction reaches that particular state. This is used to make initial predictions about a transaction, and to refine and validate those predictions as it executes. The table's values are derived from the probability distributions of the state transitions inherent in a Markov model, but are pre-computed in order to avoid having to perform an expensive traversal of the model for each transaction. This step is optional but reduces the on-line computing time for each transaction by an average of 24%, which is important for short-lived transactions. As shown in Fig. 5.4, a probability table contains two types of estimates. The first type are global predictions on (1) the probability that the transaction's future queries will execute on the same partition as where its control code is executing (**OP1**) and (2) the probability that the transaction will abort (**OP3**). For each partition in the cluster, the table also includes the probability that a transaction will execute a query that either reads or writes data at that partition (**OP2**), or conversely whether a transaction is finished at that partition (**OP4**).

5.2.2 Model Generation

A stored procedure's Markov model is generated in two parts. In the first part, called the *construction phase*, we create all known execution states from the workload trace. Next, in the *processing phase*, we traverse the model and calculate its probability distributions. We discuss adding new states at run time in Section 5.3.4.

Construction Phase: A new model for a transaction initially contains no edges and the three vertices for the `begin`, `commit`, and `abort` states. For each transaction record in the workload trace, we estimate the partitions accessed by its queries using the DBMS's internal API for the target cluster configuration [55]. We then traverse the corresponding path in the model, adding vertices and edges where appropriate. After all queries in the transaction have been processed, the last vertex in the transaction's path is connected to one of the terminal states. At the end of this phase, all of the initial execution states and edges have been created.

Processing Phase: In terms of the model, an edge's probability represents the likelihood that a transaction at the parent vertex will transition along the edge to the child vertex. In terms of the transaction, this is the probability that a transaction that has reached the parent state will execute the child's query next. The processing phase visits each vertex in the model, assigning probabilities to each outgoing edge. The probability is computed as the number of times an edge was visited divided by the total number of times the vertex was reached in the construction phase.

After the edge probabilities are calculated, we then pre-compute the vertex probability tables. A vertex's probability table is based on its children's tables weighted by their edge probabilities. The first step is, therefore, to initialize the default probabilities in the terminal states: all of the partition-specific probabilities at the `commit` vertex and the global abort probability at the `abort` vertex are both set to one. Then to calculate the tables for the remaining vertices, we traverse the model in ascending order based on the length of the longest path from each vertex to either the `commit` or `abort` vertex. Traversing the graph in this manner ensures that a vertex's table is only calculated after the tables for all of its children have been calculated. If the query at a vertex reads or writes data at a particular partition, then the corresponding entry in that vertex's probability table for that partition is set to one and the finish probability is set to zero. For those partitions not accessed at a state, then the read/write/finish probabilities are the sum of their children vertices' table entries at that partition weighted on the edge probabilities to each of those child vertices.

5.3 Predictive Framework

Given this definition of our Markov models, we now present *Houdini* , a framework for "magically" predicting the actions of transactions at run time. Such a framework can be embedded in a DBMS to enable it to automatically optimize its workload. Houdini's functionalities are designed to be autonomous, and thus do not require human intervention to maintain once it is deployed.

As shown in Fig. 5.5, Houdini is deployed on each node in the cluster and is provided with all of the Markov models generated off-line for the application's stored procedures. When a transaction request arrives at a node, the DBMS passes the request (i.e., procedure name and input parameters) to Houdini, which then generates an *initial estimate* of the transaction's execution path. This path represents the execution states that the transaction will likely reach in the Markov model for that procedure. From this initial path, Houdini informs the DBMS which of the optimizations described in Section 5.1 to enable for that request.

Determining the initial properties of a transaction before it executes is the critical component of our work. We first describe a technique for mapping the procedure input parameters to query parameters so that we can predict what partitions queries will access. We then describe how to construct the initial path in our Markov models using these parameter mappings and how Houdini uses it to select which optimizations to enable. Lastly, we discuss how Houdini checks whether the initial path matches what the transaction does and makes adjustments in the DBMS.

5.3.1 Parameter Mappings

We first observe that for most transactions in OLTP workloads, the set of partitions that each query will access is dependent on its input parameters and the database's current state [211]. A corollary to this is that the query

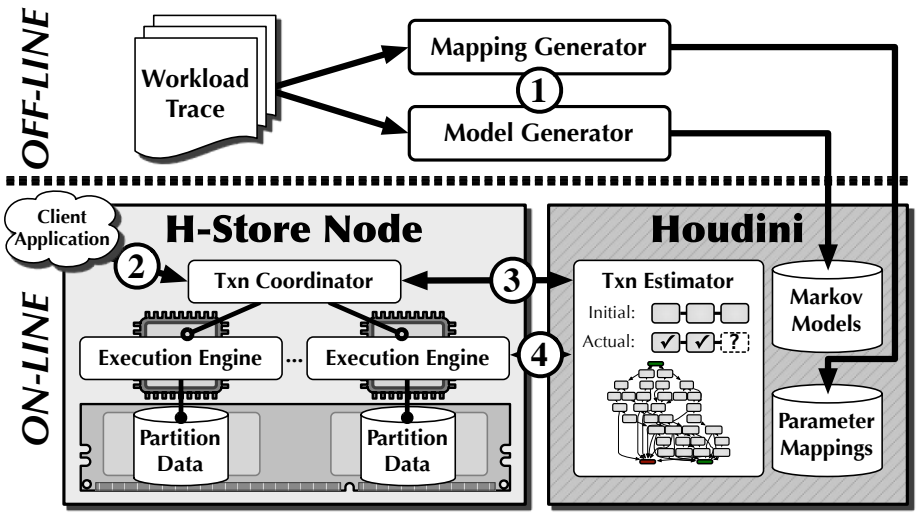


Figure 5.5: An overview of the Houdini predictive framework: (1) at initialization time, Houdini generates the Markov models and parameter mappings using a workload trace; (2) at run time, the client sends transaction requests to the DBMS's transaction coordinator; (3) the DBMS passes this request to Houdini, which generates an initial path estimate and selects optimizations; (4) Houdini monitors the transaction as it executes and provides updates to the DBMS.

parameters that are used in predicates on tables' partitioning attributes are often provided as procedure input parameters, and therefore they are not dependent on the output of earlier queries in the transaction. For example, the first input parameter to Fig. 5.2 is the warehouse id (`w_id`) that is used as an input parameter for almost all of the queries in `NewOrder`. Given this, for those queries whose input parameters that are "linked" to procedure parameters, we can determine what partitions the queries will access using the values of the procedure parameters at run time. Although procedures that do not follow this rule do exist, in our experience they are the exception in OLTP applications or are the byproduct of poor application design.

To capture such relationships, we use a data structure called a *parameter mapping* that is derived from the sample workload trace. A procedure's parameter mapping identifies (1) the procedure input parameters that are also used as query input parameters and (2) the input parameters for one query that are also used as the input parameters for other queries. We use a dynamic analysis technique to derive mappings from a sample workload trace. One could also use static analysis techniques, such as symbolic evaluation or taint checking, but these approaches would still need to be combined with traces using dataflow analysis since a transaction's execution path could be dependent on the state of the database.

To create a new mapping for a procedure, we examine each transaction record for that procedure in the workload and compare its procedure input parameters with all of the input parameters for each query executed in that transaction. For each unique pairwise combination of procedure parameters and query parameters, we count the number of times that the two parameters had the same value in a transaction. After processing all of the records in this manner, we then calculate the *mapping coefficient* for all parameter pairs as the number of times that the values for that pair were the same divided by the number of comparisons performed. As shown in the example in Fig. 5.6, the first procedure parameter has the same value as the first query parameter

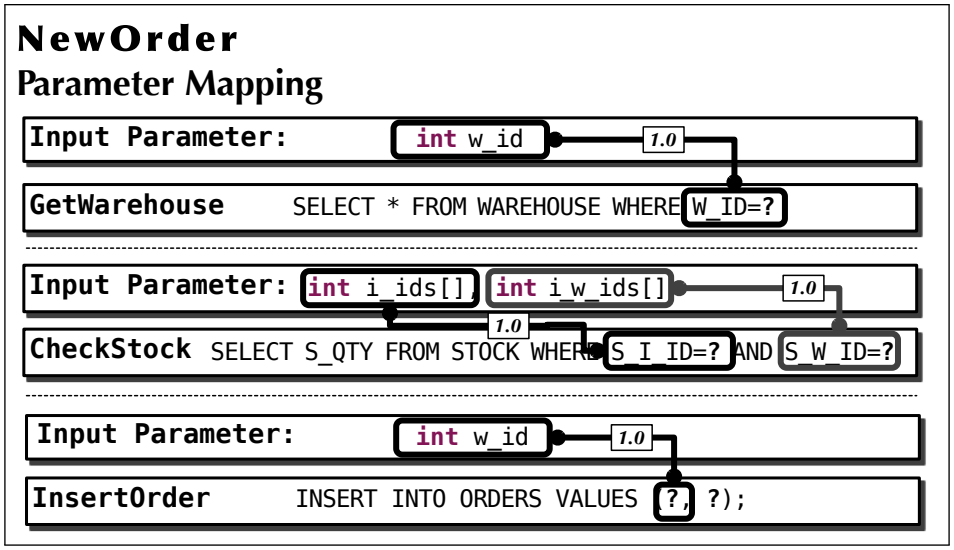


Figure 5.6: A parameter mapping for the `NewOrder` procedure.

for `GetWarehouse` (i.e., the mapping coefficient is equal to one), and thus we infer that they are the same variable in the procedure's control code. We apply this same technique to the other queries and map their input parameters as well. Note that there is not a mapping from `W_NEXT_O_ID` to the input parameters for `InsertOrder` because we do not keep track of the output values produced by queries.

A parameter mapping also supports transactions where the same query is executed multiple times and when the stored procedure has non-scalar input parameters. If a query is executed multiple times in the same transaction, then each invocation is considered a unique query. Likewise, if a procedure input parameter is an array, then each element of that array is treated as a unique parameter. From the mapping in Fig. 5.6, we identify that the n-th element of the `i_ids` array is linked to the third parameter of the n-th invocation of `InsertOrdLine` in Fig. 5.2. For each element in a procedure parameter array, we compare it with all of the query parameters within the current transaction just as before. The coefficients for multiple query instances or array parameters are aggregated into a single value using their geometric mean.

We remove false positives by discarding any mapping coefficients that are below a threshold; these occur when parameters randomly have the same values or when the control code contains a conditional block that modifies the input parameter. We found empirically that coefficients greater than 0.9 seem to all give the same result for the workloads that we investigated.

5.3.2 Initial Execution Path Estimation

Now with the procedure parameter mappings, Houdini constructs the initial execution path estimate in the Markov models for each new transaction request that arrives at the DBMS.

To generate a path estimate for a transaction, we first enumerate all of the successor states to the `begin` state and construct the set of candidate queries. We then estimate which partitions these candidates queries will access using the procedure's parameter mapping. This determines whether transitioning from the current state to the state represented by these queries (and the set of partitions that they access) is valid. A state

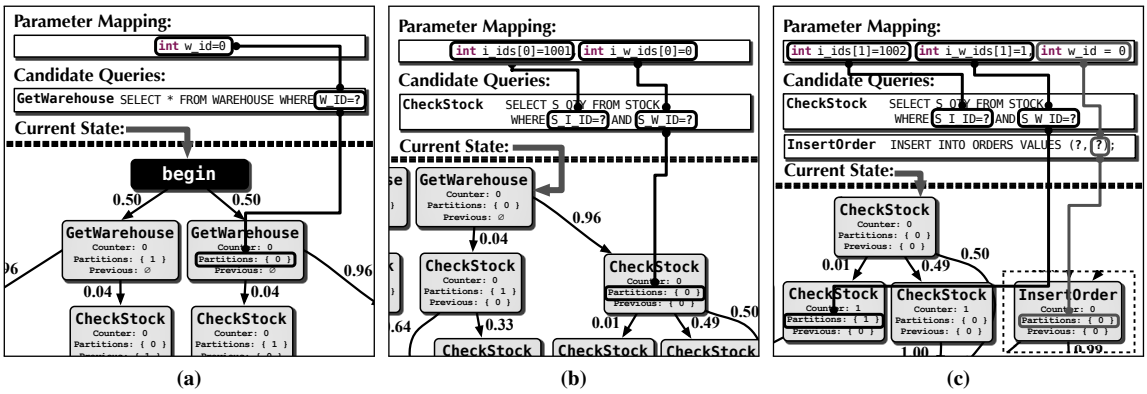


Figure 5.7: An example of generating the initial execution path estimate for a `NewOrder` invocation. As shown in the trace record in Fig. 5.6, the procedure parameters in this example are (`w_id=0`, `i_ids=[1001,1002]`, `w_i_ids=[0,1]`, `i_qtys=[2,7]`).

transition is *valid* for a transaction if (1) we can determine all the query parameters needed for calculating the partitions accessed by that state's query and (2) the next state's set of previously accessed partitions contains all the partitions accessed by the transaction up to this point. For those transitions that are valid, we choose the one with the greatest edge probability, append this state to the initial path estimate, and then repeat the process.

We now illustrate these steps using the `NewOrder` Markov model shown in Fig. 5.3. As shown in Fig. 5.7a, the only candidate query when a transaction starts is `GetWarehouse`. Using the parameter mapping shown in Fig. 5.6, we identify that `GetWarehouse`'s first parameter is mapped to the procedure's first parameter (`w_id`). Therefore, we can compute which partitions `GetWarehouse` accesses because we know with a high-degree of certainty the value of its only input parameter is the same as the value of `w_id`. We then select the next state as the particular `GetWarehouse` state that accesses the same partitions as was estimated. This process is repeated in the next step in Fig. 5.7b: the candidate query set contains only `CheckStock`, so we again use the mapping to get the values of the procedure parameters that are used for that query and compute which partitions that the query accesses. We then select the next state in the path as the one that represents the first invocation of `CheckStock` that accesses these partitions and also has the correct previously accessed partitions for the transaction.

The technique described above works well when either the procedure's control code is linear or all the pertinent query parameter are mappable to procedure parameters, whereupon the Markov model is essentially a state machine. But many procedures contain conditional branches, and thus it is not always possible to resolve which state is next simply by estimating partitions. An example of this is shown in Fig. 5.7c. There are two choices for which query that the transaction could execute next: (1) the second invocation of `CheckStock` or (2) the first invocation of `InsertOrder`. Both transitions are valid if the size of the `i_ids` procedure input parameter array is greater than one. If the size of this array was one, then Houdini would infer that the transaction could never execute `CheckStock` a second time. When such uncertainty arises, we chose the edge with the greater weight.

The path estimate is complete once the transaction transitions to either the `commit` or `abort` state.

5.3.3 Initial Optimizations Selection

Using a transaction's initial path estimate, Houdini chooses which optimizations the DBMS should enable when it executes the transaction. We now describe how Houdini selects these optimizations.

For each potential optimization, we calculate a *confidence coefficient* that denotes how likely that it is correct. This coefficient is based on the probabilities of the edges selected in the transaction's initial path estimate. Houdini prunes estimations if their corresponding confidence is less than a certain threshold. Setting this threshold too high creates false negatives, preventing the DBMS from enabling valid optimizations. Conversely, setting this threshold too low creates false positives, causing the DBMS to enable certain optimizations for transactions that turn out to be incorrect and therefore it will have to rollback work. We explore the sensitivity of this threshold in our evaluation in Section 5.5.4.

OP1: Houdini counts each time that a partition is accessed by a query in the transaction's initial path estimate. The partition that is accessed the most is selected as the transaction's base partition.

OP2: Similarly, the set of partitions that the transaction needs (and therefore the DBMS should lock) is the based on the execution states in the initial path estimate. The probability that a partition is accessed is the confidence coefficient of the edges in the initial path up to the first vertex that accesses that partition.

OP3: Because multi-partition and speculatively executed transactions can be aborted as a result of other transactions in the system, these transactions are always executed with undo logging enabled. Thus, Houdini will only determine which non-speculative single-partition transactions can be executed without undo buffers. Houdini is more cautious when estimating whether transactions could abort because unlike the other optimizations, it will be expensive to recover if it is wrong. To avoid this, we use the greatest abort probability in the all of the tables in the initial path estimate. That is, the probability that the transaction will abort is the largest abort probability value in all of the states' tables.

5.3.4 Optimization Updates

After creating the initial path and optimization estimates for a transaction, Houdini provides this information to the DBMS. The transaction is then queued for execution at the current node or redirected based on the estimate. Once the transaction starts, Houdini tracks its execution and constructs the path of execution states that the transaction enters in its stored procedure's model. At each state, Houdini (1) determines whether the transaction has deviated from the initial path estimate and (2) derives new information based on the transaction's current state. If the transaction reaches a state that does not exist in the model, then a new vertex is added as a placeholder; no further information can be derived about that state until Houdini recomputes the model's probabilities (Section 5.3.5). Otherwise, Houdini uses the current state to provide updates to the DBMS's transaction coordinator:

OP3: Houdini uses the pre-calculated probability tables to check whether a single-partition transaction has reached a point in its control code that will never abort (i.e., there is no path from the current state to the

abort state). When this occurs, the DBMS disables undo logging for the remainder of the transaction's execution.

OP4: Houdini also uses the probability tables to determine whether a distributed transaction is finished with partitions. If the finish probability for a particular partition is above the confidence threshold, then Houdini informs the DBMS that the transaction no longer needs that partition. This allows the DBMS to send the early prepare message [193] and speculatively execute transactions at these partitions [37, 131]. If the transaction was read-only at a partition, then it commits immediately and the DBMS begins to execute other transactions on that partition. Otherwise, the speculative transaction waits until the distributed transaction finishes.

5.3.5 Model Maintenance

The probability that a transaction transitions from one state to another is based on static properties of the sample workload trace that was used to generate the models. If an application's workload shifts, then the models may no longer represent the current behavioral state of that application. For example, if previous NewOrder transactions in the trace only inserted two items but now incoming requests have three or more, then Houdini will incorrectly choose initial paths that only executed the CheckStock query twice. Houdini can identify when the workload has changed [119] and to adjust to these changes without having to re-create the models. This occurs on-line without stopping the system; new models only need to be generated off-line when the database's partitioning scheme changes or when the procedure's control code is modified.

Houdini determines whether a model is no longer accurate by measuring how often it chooses a state transition for transactions that does not match the expected edge probability distribution. As a transaction executes, Houdini constructs its actual execution path in the model and increments internal counters whenever the transaction "visits" an edge. As long as the distribution of the transitions from each vertex is within some threshold of the original probabilities in the Markov model, then Houdini infers that the model is still in sync with the application. If the distribution no longer matches the model's expectations, then Houdini recalculates the edge and vertex probabilities based on the edge counters. Since this is an inexpensive operation ($\leq$ 5 ms), our current implementation uses a threshold of 75% accuracy before Houdini recomputes the probabilities. We defer the exploration of more robust techniques as future work, such as a sliding window that only includes recent transactions for fast changing workloads.

5.3.6 Limitations

There are three ways that Houdini may fail to improve the throughput of the DBMS. The first case is if the overhead from calculating the initial path estimate negates the optimizations' performance gains. This can occur if a model is very wide (i.e., many transition possibilities per state) or very long (i.e., many queries executed per transaction). For the latter, the limit is approximately 175-200 queries per transaction in our current implementation. It simply takes too long for Houdini to traverse the model for these transactions and compute the partitions that could be accessed at each state. Pre-computing initial path estimates for stored procedures that are always single-partition would alleviate this problem to some extent, but it is not applicable for procedures that are distributed only some of the time since Houdini needs the path estimate to

determine what partitions will be accessed. We note, however, that procedures that execute many queries and touch a large portion of the database are not the main focus of high-performance OLTP systems. They are often "clean-up" transactions that are executed at periodic intervals to perform maintenance operations on the entire database and thus are unlikely to benefit from the optimizations enabled by Houdini.

Additionally, storing all of the execution states for a stored procedure in a single "global" Markov model can be difficult to scale for large clusters. The total number of states per model is combinatorial for procedures like `NewOrder` that access combinations of partitions, most of which are unreachable based on where the transaction's control code is executing. For example, a transaction executing at a particular partition can only reach just one of the `GetWarehouse` states in Fig. 5.3b and based on which one that is, other states can never be reached. These global models are also problematic on multi-core nodes, since Houdini must either use separate copies of the models for each execution thread, or use locks to avoid consistency issues when it updates the models.

The last type of limitation that can hinder DBMS performance is if the models are unable to accurately predict what a transaction will do, causing the DBMS to make wrong decisions and possibly have to redo work. As an example of this, consider a `NewOrder` request that has two items to insert from different warehouses (i.e., partitions). If Houdini uses the Markov model in Fig. 5.3 to predict this transaction's initial path, then it would not select the correct execution state from the choices shown in Fig. 5.7c. This is because the second invocation of `CheckStock` and the `InsertOrder` query are both valid states; the length of the length of the warehouse id array (`i_w_ids`) is greater than one, and thus the transaction could potentially execute either query. As described in Section 5.3.2, when such uncertainty arises, we choose the next state transition based on edge with the greatest probability. This is still insufficient, however, since the probability of the transition that the transaction will actually take is less than the other potential transition. The Markov model in Fig. 5.3 does not capture the fact that the number of `CheckStock` queries corresponds to the length of the `i_w_ids` array. This is problematic in our example because the query that Houdini failed to predict in the model accesses a partition that is different than the ones from the transaction's previous queries. This means that Houdini will have incorrectly predicted that the transaction is single-partitioned.

5.4 Model Partitioning

Given these limitations, we now describe how Houdini automatically partitions the Markov models for a given application to improve their prediction efficacy and scalability. Houdini clusters the transactions for each procedure in the sample workload trace based on salient attributes of its input parameters. This allows us to capture certain nuances of the transactions, such as variability in the size of input parameter arrays. As shown in Fig. 5.8, we generate models for each of these clusters and support them with a decision tree that allows Houdini to quickly select the right model to use for each incoming transaction request at run time.

Dividing the models in the manner that we now describe is a well-known and effective technique from the machine learning and optimization communities [132, 227].

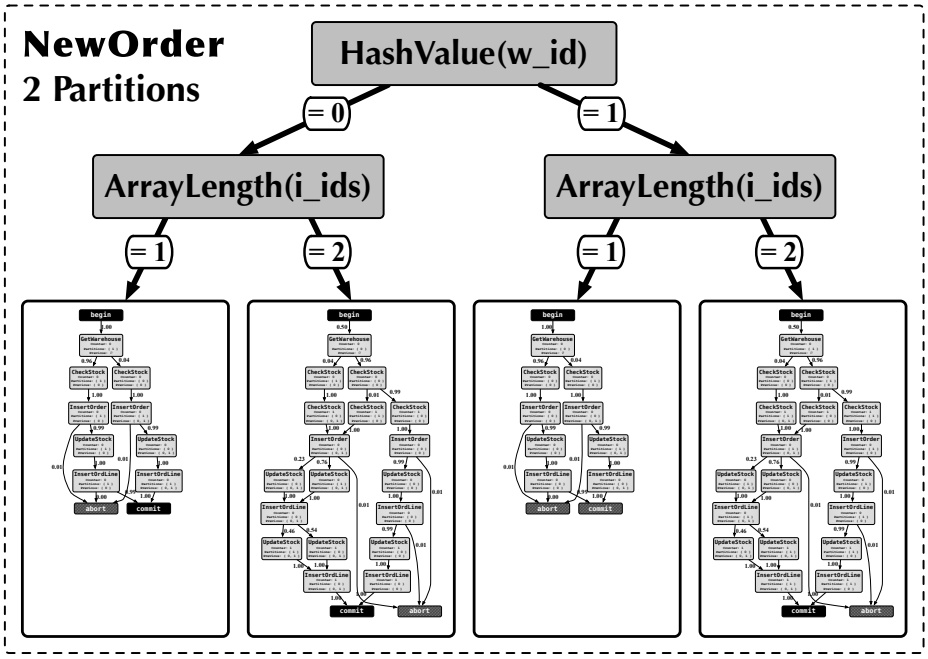


Figure 5.8: A partitioned set of `NewOrder` Markov models. The decision tree above the models divides transactions by the hash value of the first procedure parameter and the length of the array of the second procedure parameter. The detail of the models in the above figure is not relevant other than to note that they are less complex than the global model for the same procedure shown in Fig. 5.3.

Feature Category	Description
NORMALIZEDVALUE(x)	The normalized value of the parameter x.
HASHVALUE(x)	The hash value of the parameter x.
ISNULL(x)	Whether the value of the parameter x is null.
ARRAYLENGTH(x)	The length of the array parameter x.
ARRAYALLSAMEHASH(x)	Whether all elements of the array parameter x hash to the value.
EQUALSBASEPARTITION(x)	Whether parameter x hashes to the same value as the txn's base partition.

Table 5.1: The list of feature categories that are extracted from the stored procedure input parameters for each transaction trace record. These features are used when sub-dividing the models for each stored procedure to improve scalability and prediction accuracy.

Feature Instance	Value	Feature Instance	Value
HASHVALUE(`w_id`)	0	ARRAYLENGTH(`w_id`)	*null*
HASHVALUE(`i_ids`)	*null*	ARRAYLENGTH(`i_ids`)	2
HASHVALUE(`i_w_id`)	*null*	ARRAYLENGTH(`i_w_ids`)	2
HASHVALUE(`i_qtys`)	*null*	ARRAYLENGTH(`i_qtys`)	2

Table 5.2: The feature vector extracted from the transaction example in Fig. 5.7. The value for the ARRAYLENGTH(`w_id`) feature is null because the `w_id` procedure parameter in Fig. 5.2 is not an array.

5.4.1 Clustering

The goal of the clustering process is to group transactions together based on their features in such a way that the Markov models for each cluster more accurately represent the transactions. We define a *feature* in

this context as an attribute that is derived from a transaction's stored procedure input parameters [103]. For example, one feature could be the length of the array for one particular parameter, while another could be whether all of the values in that array are the same. Table 5.1 shows the different categories of features that are extracted from transaction records. A *feature vector* is a list of values for these features that are extracted from each transaction trace record in the sample workload. Each transaction's feature vector contains one value per input parameter per category. An example of a feature vector is shown in Table 5.2

After extracting the feature vectors for each of the transaction records in the workload trace, we then employ a machine learning toolkit to cluster the transactions of each procedure based on these vectors [107]. We use the *expected maximization* clustering algorithm, as it does not require one to specify the number of clusters beforehand. The transaction records are each assigned to a cluster by this algorithm and then we train a new Markov model that is specific for each cluster using these records. For example, if we clustered the `NewOrder` transactions based on the length of the `i_w_ids` input parameter, then the number of `CheckStock` invocations for all transactions in each cluster will be the same.

5.4.2 Feed-Forward Selection

The problem with the above clustering approach is that it is decoupled from Houdini's ability to predict a transaction's properties accurately using the models; that is, the clustering algorithm may choose clusters for a stored procedure based on features that do not improve the accuracy of the models' predictions compared to the single non-clustered model. Therefore, we need to determine which set of features are relevant for each procedure in order to cluster the transactions properly. Enumerating the power set of features with a brute-force search to evaluate the accuracy of all feature combinations is not feasible, since the amount of time needed to find the optimal feature set is exponential. This would simply take too long for applications either with a large number of stored procedures or with stored procedures that have many input parameters.

We instead use a greedy algorithm called *feed-forward selection* as a faster alternative [103, 132]. This algorithm first iterates all unique feature combinations for small set sizes and then constructs larger sets using only those features that were in the smaller sets that improve the predictions. In each round r, we create all sets of features of size r and measure how well they predict the initial execution paths of transactions. After each round, we sort the feature sets in ascending order and select the features in the top 10% sets with the best accuracy. We repeat the process in the next round using sets of size $r + 1$. The search stops when at the end of a round the algorithm fails to find at least one feature set that produces clustered models with better prediction accuracy than the best feature set found in the previous rounds.

To begin, we first split the sample workload for the target stored procedure into three disjoint segments, called the *training workset* (30%), the *validation workset* (30%), and the *testing workset* (40%) [132]. Then we enumerate the power set of features for the current round (e.g., if there n features, then the initial round will have n one-element sets). For each feature set in the round, we seed the clustering algorithm on that set using the training workset. We then use the seeded clusterer to divide the transaction records in the validation workset and generate the Markov models for each cluster using the same method described in Section 5.2.2.

Now with a separate Markov model per cluster for a particular feature set, we estimate the accuracy of the clustered models using the remaining records in the testing workset. For each of these transaction records, we generate an initial path estimate using Houdini just as if it was a new transaction request and then simulate the

transaction executing in the system by generating the "actual" execution path of the transaction. We measure the accuracy of these initial path estimates not only based on whether it has the same execution states as the actual path, but also based on whether Houdini correctly generates transaction updates.

The accuracy for each initial path estimate is based on the optimizations defined in Section 5.1. The penalty for incorrectly predicting that a single-partition transaction will not abort is infinite, since it puts the database in an unrecoverable state. The total accuracy measurement for each feature set is the sum of these penalties for all transactions in the testing workset.

5.4.3 Run Time Decision Tree

After the search terminates, we use the feature set with the lowest cost (i.e., most accurately models the transactions for the target stored procedure) to generate a decision tree for the models using the C4.5 classifier algorithm from the same machine learning toolkit [107]. When a new transaction request arrives at the DBMS at run time, Houdini extracts the feature vector for the transaction and traverses this decision tree to select which Markov model to use for that request. For example, the Markov models shown Fig. 5.8 for the `NewOrder` stored procedure are clustered on the value of the `w_id` parameter and the length of the `i_w_ids` array. The models in the leaf nodes of the tree are specific to these features. This mitigates the scaling, concurrency, and accuracy problems from using a single model per procedure.

5.5 Experimental Evaluation

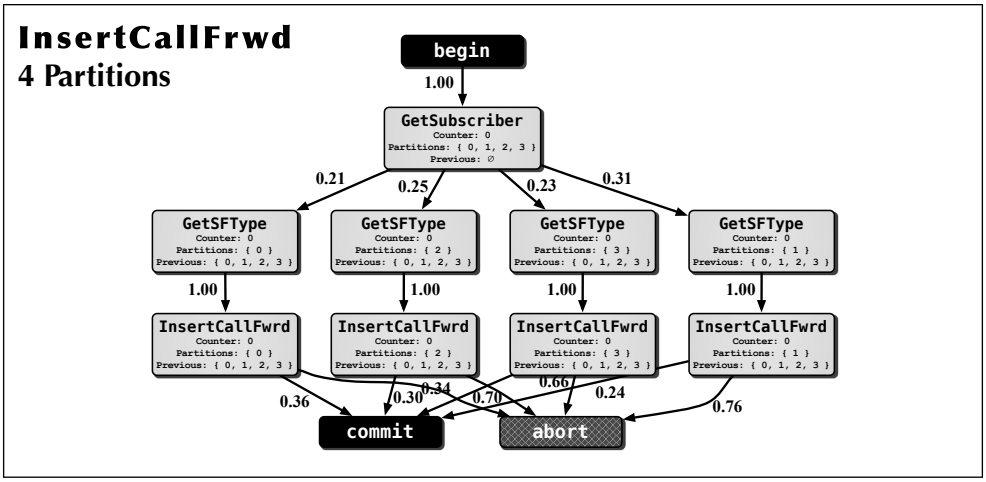
We have integrated our modeling algorithms and prediction framework in the H-Store system and now present an evaluation of its usefulness. We use three OLTP benchmarks that have differing levels of complexity in their workloads: TATP, TPC-C, and AuctionMark (cf. Appendix A). We assume that the databases for each benchmark are partitioned in such way that it maximizes the number of single-partition transactions [181]. For each benchmark, we generate sample workload traces of 100,000 transactions collected over a simulated one hour period.

All of the experiments measuring throughput were conducted on a cluster at the University of Wisconsin-Madison. Each node has a single 2.4GHz Intel Core 2 Duo processor with 4GB RAM.

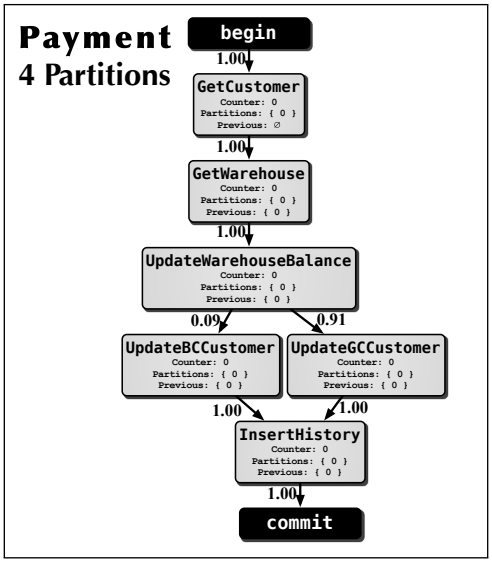
5.5.1 Model Accuracy

We first calculated the off-line accuracy of the optimization estimates generated by Houdini for a simulated cluster of 16 partitions. The accuracy of an estimate is based on whether Houdini (1) identifies the optimizations at the correct moment in the transaction's execution (e.g., disabling undo logging at the right time – **OP3**), (2) does not cause the DBMS to perform unnecessary work (e.g., locking partitions that are never used – **OP1**, **OP2**), and (3) does not cause the transaction to be aborted and restarted (e.g., accessing a partition after it was deemed finished – **OP4**). For each procedure, we generate a single "global" model and a set of "partitioned" models using the first 50,000 transaction records from the sample workloads. We then use Houdini to estimate optimizations for the remaining 50,000 transactions. We reset the models after each estimation so as to not learn about new execution states, which would mask any deficiencies.

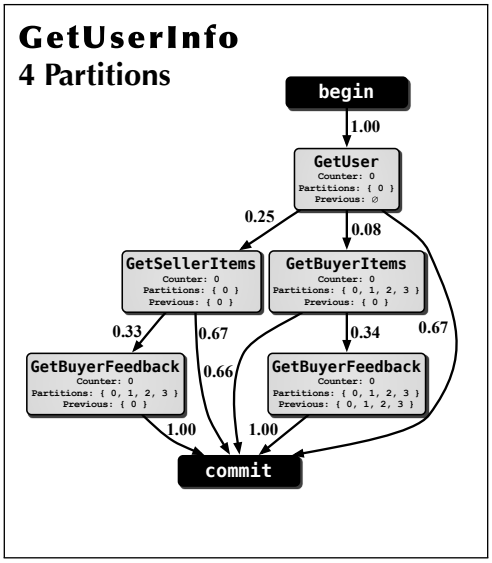
The results in Table 5.3 show that the global Markov models enable accurate path estimates for 91.0% of



(a) TATP



(b) TPC-C



(c) AuctionMark

Figure 5.9: Markov models for select stored procedures from the TATP, TPC-C, and AuctionMark OLTP benchmarks used in our evaluation in Section 5.5.

the transactions evaluated, while the partitioned models improved the accuracy rate to 93.4%. Although Houdini fails to select the base partition (**OP1**) for 5% of TATP's transactions, they are all distributed transactions that either update every partition (i.e., the base partition does not matter) or access a single partition based on the result of a multi-partition query (i.e., the best base partition depends on the state of the database). The accuracy for TPC-C is nearly perfect in the partitioned models for **OP1-3**, but that it can miss that a transaction is with finished a partition (**OP4**). The accuracy for AuctionMark transactions is also high, except for the two procedures with conditional branches. Houdini never mispredicts that a transaction will not abort for any benchmark, but it does miss a small number of transactions ($<$1%) where it could have disabled undo logging (**OP3**).

		TATP	TPC-C	AuctionMark
OP1	Global	95.0%	94.8%	94.9%
	Partitioned	94.9%	99.9%	94.7%
OP2	Global	98.9%	90.9%	90.7%
	Partitioned	100%	99.0%	95.4%
OP3	Global	100%	100%	100%
	Partitioned	100%	100%	100%
OP4	Global	99.5%	100%	100%
	Partitioned	99.5%	95.8%	99.9%
Total	Global	**94.9%**	**93.8%**	**85.6%**
	Partitioned	**94.9%**	**95.0%**	**90.2%**

Table 5.3: Measurements of the global and partitioned Markov models' accuracy in predicting the execution properties of transactions.

5.5.2 Estimation Overhead

Next, we measured the overhead of using Houdini to estimate the optimizations at run time. We implemented a profiler [112] that records the amount of time H-Store spends for each transaction (1) estimating the initial execution path and updates, (2) executing its control code and queries, (3) planning its execution, (4) coordinating its execution, and (5) miscellaneous setup operations. We executed the benchmarks on a 16-partition H-Store cluster and report the average time for each of these measurements. Profiling begins when a request arrives at a node and then stops when the result is sent back to the client. We use the partitioned models so that the cost of traversing the decision tree is included in the measurements.

The results in Fig. 5.10 show that only an average of 5.8% of the transactions' total execution time is spent in Houdini. This time is shared equally between estimating the initial path versus calculating updates. All procedures with an overhead greater than 15% are short-lived single-partitioned transactions. For example, 46.5% of AuctionMark `NewComment`'s execution time is spent selecting optimizations, but it is the shortest transaction (i.e., average execution time is just 0.29 ms). Although we do not discuss such techniques in this dissertation, Houdini can completely avoid this if it caches the estimations for any non-abortable, always single-partition transactions.

5.5.3 Transaction Throughput

We next measured the throughput of H-Store when deployed with Houdini. We execute each benchmark using five different cluster sizes, with two partitions assigned per node. Transaction requests are submitted from multiple client processes running on separate machines in the cluster. We use four client threads per partition to ensure that the workload queues at each node are always full. We execute each benchmark three times per cluster size and report the average throughput of these trials. In each trial, the DBMS is allowed to "warm-up" for 60 seconds and then the throughput is measured after five minutes. As H-Store executes, we record the percentage of transactions for each procedure where Houdini successfully selected an optimization. Note that this is different than the accuracy measurements shown Table 5.3, because Houdini now must consider the run time state of the DBMS (e.g., it cannot disable undo logging for speculative transactions).

We executed the benchmarks with Houdini first using the global Markov models and then again using the

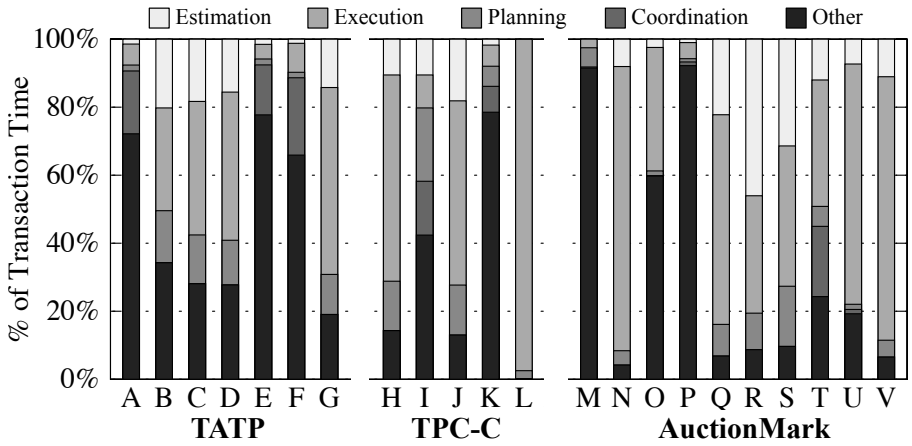


Figure 5.10: Relative measurements of the time spent for each transaction (1) estimating optimizations, (2) executing, (3) planning, (4) coordinating its execution, and (5) other setup operations.

partitioned models. We allow Houdini to "learn" about new execution states in the models in the warm-up period, and then recompute the probabilities before running the measured workload. For each new transaction request, Houdini generates the initial path estimate and determines whether the request needs to be redirected to a different node (**OP1**) and can be executed with undo logging (**OP3**). As the transaction executes, Houdini checks whether it is finished with other partitions (**OP4**) or no longer needs undo logging (**OP3**). Any transaction that attempts to access a partition that Houdini failed to predict (**OP2**) is aborted and restarted as a multi-partition transaction that locks all partitions.

To compare how H-Store performs without Houdini, we also executed the benchmarks using DB2-style transaction redirects [59]. When operating in this mode, the DBMS first executes every request as single-partition transaction at a random partition on the node where the request originally arrived. If a transaction attempts to access a partition that is different than the one it was assigned to, then it is aborted and redirected to the correct node. If the transaction attempts to access multiple partitions, none of which are at the node where it is currently executing at, then it is redirected to the partition that it requested the most and is executed as a multi-partition transaction (with random tiebreakers). Because the DBMS has no way to infer the transaction properties without Houdini, it cannot use the other optimizations.

TATP: The results in Fig. 5.11a show that there is a 26% throughput improvement when using the partitioned models with Houdini. This is mainly attributable to Houdini identifying the best base partition for 82% of TATP's workload that is singled-partitioned (**OP1**, **OP2**). The other 18% first execute a broadcast query on all partitions, thus locking a subset of the partitions is not possible (**OP2**). Subsequent queries in these transactions only access a single partition based on the result of the first query. This also makes it impossible to select the correct base partition for each transaction (**OP1**), since the Houdini cannot know which partition will be needed after the broadcast query. Thus, without the early prepare optimization, all of the other partitions would remain idle (**OP4**), albeit for just a short amount of time. An example of a Markov model for this access pattern is shown in Fig. 5.9a. Additionally, as shown in Table 5.4, Houdini disables

		Procedure	OP1	OP2	OP3	OP4	Estimate
TATP	A	DeleteCallFwrd	-	100%	-	-	0.02 ms
	B	GetAccessData	98.5%	100%	64.8%	33.7%	0.01 ms
	C	GetNewDest	100%	100%	66.4%	33.6%	0.01 ms
	D	GetSubscriber	98.9%	100%	64.9%	34.1%	0.01 ms
	E	InsertCallFwrd	-	100%	-	-	0.04 ms
	F	UpdateLocation	-	100%	-	-	0.01 ms
	G	UpdateSubscriber	100%	100%	-	53.2%	0.02 ms
TPC-C	H	Delivery	100%	100%	78.6%	22.4%	4.23 ms
	I	NewOrder	99.5%	93.2%	72.5%	19.6%	0.43 ms
	J	OrderStatus	100%	100%	89.6%	85.3%	0.05 ms
	K	Payment	99.1%	99.7%	60.6%	16.4%	0.08 ms
	L	StockLevel	99.2%	100%	46.7%	22.0%	0.05 ms
AuctionMark	M	CheckWinningBids	-	-	-	-	-
	N	GetItem	100%	100%	89.0%	11.0%	0.04 ms
	O	GetUserInfo	99.9%	100%	75.3%	8.4%	0.05 ms
	P	GetWatchedItems	100%	100%	-	-	0.04 ms
	Q	NewBid	100%	100%	83.2%	13.3%	0.26 ms
	R	NewComment	99.5%	100%	44.6%	11.3%	0.13 ms
	S	NewItem	100%	100%	95.9%	4.1%	0.20 ms
	T	NewPurchase	99.0%	100%	46.4%	11.1%	0.12 ms
	U	PostAuction	-	55.0%	-	16.7%	0.32 ms
	V	UpdateItem	100%	100%	90.5%	9.5%	0.04 ms

Table 5.4: The percentage of transactions that Houdini successfully enabled one of the four optimizations. In the case of **OP4**, the measurement represents how many transactions were speculatively executed as a result of the early prepare optimization. The rightmost column contains the average amount of time that Houdini spent calculating the initial optimization estimates and updates at run time.

undo logging for 57.3% of TATP's transactions (**OP3**), but this has negligible impact since the transactions execute only 1-3 queries or are read-only.

The throughput of global models is 4.5% slower on average than the partitioned models due to lock contention in Houdini.

TPC-C: This benchmark's results in Fig. 5.11b show that the "assume single-partition" method performs 6% better than Houdini for small clusters. This is because the likelihood that a transaction is already at the best base partition is greater when there are fewer partitions (**OP1**). TPC-C's procedures also execute more queries per transaction than the other benchmarks, and thus the estimations take longer to compute. As shown in Table 5.4, Houdini takes an average of 4 ms to compute estimates for `Delivery` transactions, but these transactions take over 40 ms to execute. The benefit of our techniques therefore is only evident for larger clusters: Houdini's ability to identify not only whether a transaction is distributed or not, but also which partitions it accesses improves throughput by 33.6% (**OP2**). Table 5.4 also shows that 65.3% of TPC-C's workload is executed without undo logging, most of which are after the transactions have started (**OP3**).

These results also highlight the advantage of model partitioning: the global models' size grows exponentially relative to the number of partitions, thereby increasing the time Houdini needs to traverse the model.

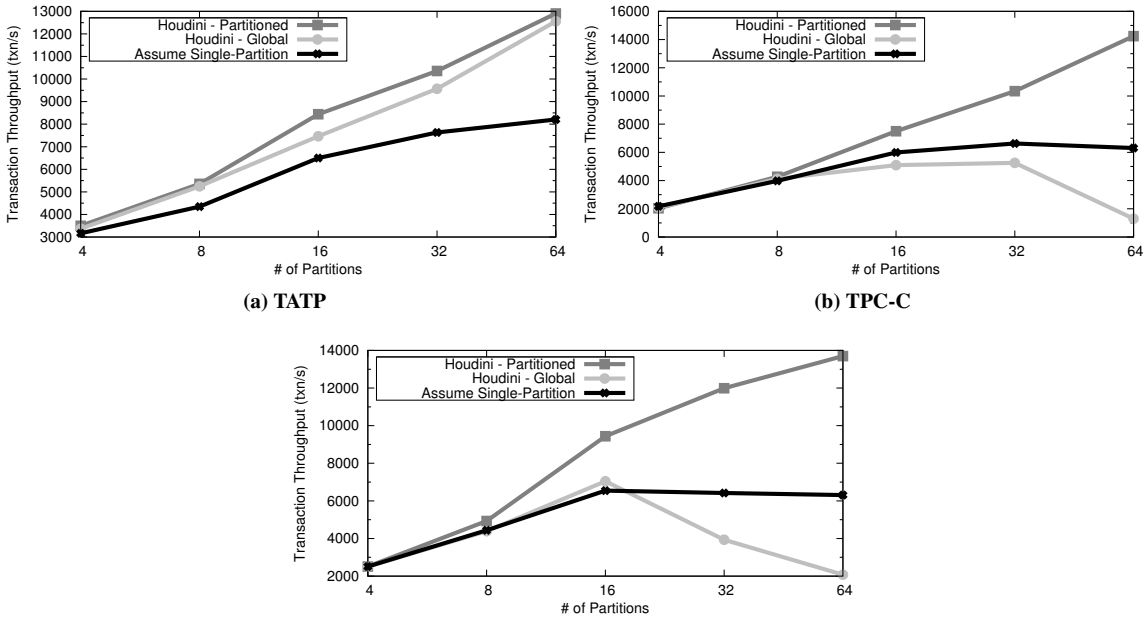


Figure 5.11: Throughput measurements of H-Store for different execution modes: (1) Houdini with partitioned Markov models; (2) Houdini with global Markov models; and (3) DB2-style transaction redirects and assuming that all partitions are single-partitioned.

The partitioned models also allow Houdini to identify the correct partitions needed for `NewOrder` and `Payment` transactions more often, resulting in fewer aborted transactions. As shown in Fig. 5.9b, partitioning `Payment`'s models creates almost linear models, which enables Houdini to easily identify when the transaction is distributed (**OP2**).

AuctionMark: The results in Fig. 5.11c show that H-Store achieves an average 47.3% performance improvement when using Houdini with partitioned models for this workload. Like TPC-C, the global models have scalability issues as the size of the cluster increases. AuctionMark mostly benefits from identifying the two partitions the lock for distributed transactions: one for the buyer and one for the seller (**OP2**). As shown in Table 5.4, Houdini identifies this optimization for 100% of the transactions. The "assume single-partition" strategy does not scale because the transactions do not access the remote partition in the first set of queries (**OP1**), thus they must always be restarted again and lock all of the partitions. Other procedures, such as `GetUserInfo` shown in Fig. 5.9c, contain conditional branches with separate single-partition and multi-partition paths. Such procedures are ideal for our model partitioning technique, but most of AuctionMark's transactions are short-lived, which means that disabling undo logging (**OP3**) and early prepare optimizations (**OP4**) only provide a modest benefit.

As explained in Section 5.3.6, we disabled Houdini for the maintenance `CheckWinningBids` requests, as it takes too long process due to the large number of queries (>175) in each transaction. Houdini also does not correctly predict the accessed partitions for 45.0% of `PostAuction` transactions (**OP2**) because their input

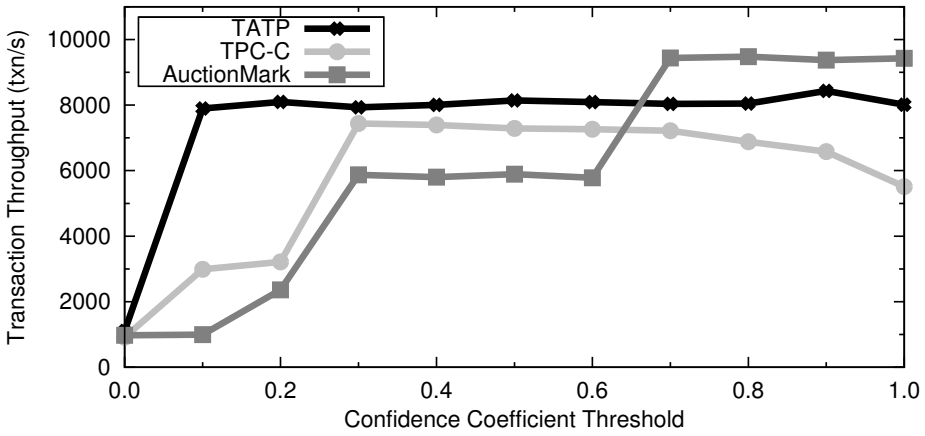


Figure 5.12: Throughput measurements of H-Store under varying estimation confidence coefficient thresholds (Section 5.3.2).

parameters are large, arbitrary length arrays, which does not work well with our model partitioning technique.

5.5.4 Confidence Sensitivity Analysis

Lastly, we measured how H-Store performs when Houdini uses different confidence coefficient thresholds to select which optimizations to enable. Recall from Section 5.3.3 that this threshold determines whether a prediction will be included for a transaction based on its confidence coefficient. We executed the benchmarks again in H-Store on a 16-partition cluster and vary the confidence threshold from zero (i.e., all estimations are permitted) to one (i.e., only the most certain estimations are permitted).

As expected, when the threshold is set to zero, the results in Fig. 5.12 show that all transactions are executed as multi-partition since Houdini predicts that each transaction will always touch all partitions. Once the threshold is $>$0.06 (i.e., $\frac{1}{16}$), the throughput for TATP remains the same because Houdini correctly identifies which partitions transactions will access (**OP1**, **OP2**) and when they are finished with them (**OP4**). For TPC-C, the throughput reaches a plateau at $>$0.3 because the number of mis-predicted **OP1** for `NewOrder` transactions is reduced from 10% to 5%, but declines slightly as the threshold approaches one because Houdini no longer selects to disable undo logging as much as it could. In the case of AuctionMark, there are two procedures with conditional branches where Houdini does predict the correct partitions (**OP1**, **OP2**) until the threshold is $>$0.33 and $>$0.66.

5.6 Conclusion

We introduced a new approach for representing the stored procedures of OLTP applications using Markov models to forecast the behavior of transactions. Such models are used to identify when the DBMS can execute a transaction using four different optimizations. From this, we then presented Houdini, a new prediction framework that uses our Markov models to estimate the execution path of future transactions. We described a method for generating these models, as well as how to partition them on certain features to improve their scalability and accuracy. To evaluate our work, we integrated Houdini into the H-Store distributed OLTP

system. The results from our experimental analysis show that our models accurately predict the execution paths of 93% of transactions in three OLTP benchmarks. We also demonstrated that our technique has only an average overhead of 5.8%, while increasing the throughput of the system by an average of 41% compared to a naïve approach. These results suggest that predicting transaction properties using Markov models could be useful for any distributed OLTP database. In future work, we will attempt to apply it to real applications and systems.

Chapter 6

Speculative Execution

In Chapter 4, we showed how the Horticulture tool generates database designs that minimize the number distributed transactions. Then in Chapter 5, we showed how the Houdini framework allows the DBMS to identify whether a transaction request should be executed as a distributed transaction or not. But despite these advancements, distributed transactions are still unavoidable in some OLTP applications. As we will now discuss, this means that a distributed DBMS needs to be able to minimize their impact.

Some distributed DBMSs employ a concurrency control scheme that allows other transactions to execute simultaneously whenever one transaction is waiting for data or synchronization messages from other nodes [30]. This allows the DBMS to mask high network latency. But even though most transactions in OLTP applications are not distributed, all transactions in the DBMS must be executed with the same concurrency control scheme to ensure correctness. But as we showed in Section 2.2, such schemes will reduce the performance of single-partition transactions on memory-resident databases due to overhead of managing concurrent transactions [112, 149]. This means that in order to support distributed transactions, these concurrency control schemes degrade the performance of the majority of an application's workload.

For many applications, a better approach is to optimize for the common case (i.e., single-partition transactions) and then have the DBMS handle distributed transactions as an exception [131, 211, 224]. Under this scheme, transactions are executed serially at their respective partitions, thereby removing all contention due to transactions accessing data at the same time. This is the approach that we use in H-Store (cf. Section 3.2). The downside of executing transactions one-at-a-time at a partition, however, is that it means the DBMS is unable to do work whenever a distributed transaction has to stall and wait for a message to arrive over the network. These stalls significantly affect performance of workloads with only a small percentage of distributed transactions [181].

Rather than always stalling in this manner, the DBMS can *speculatively execute* queued tasks in anticipation that the current distributed transaction will succeed. That is, when the DBMS does not have work for the current transaction, it executes work on behalf of other transactions that it would normally only execute after that transaction finishes. This allows the DBMS to execute tasks concurrently without the overhead of a heavy-weight scheme.

One well-known application of speculative execution is for the DBMS to begin processing queued transactions once it receives the distributed transaction's 2PC prepare notification [37, 72, 131, 180, 193]. This only accounts for a small portion of the time that transactions are stalled in distributed DBMSs. Supporting additional speculation opportunities in a system that is optimized for single-partition transactions requires knowledge about what operations each transaction will perform *before* it starts [131]. Otherwise, a speculative transaction may create an inconsistent view of the database for the stalled distributed transaction. Moreover, while this improves the system's overall throughput, it does not improve the latency of individual distributed transactions. This is also non-trivial, because again the DBMS needs to know what operations the transaction is likely to request in the future. But recent work has explored integrating machine learning techniques in the DBMS for discovering this information in OLTP applications [180].

We now present new approaches for lock-free speculative execution in distributed DBMSs. We developed a fast and efficient method, called *Hermes,* that uses probabilistic models generated from observing the behavior of previous transactions to:

1. Speculatively interleave single-partition transactions at a node whenever the distributed transaction stalls.

2. Speculatively prefetch queries on remote nodes for distributed transactions to reduce the wait time for results.

These techniques improve both the throughput and response time for OLTP applications without giving up consistency guarantees.

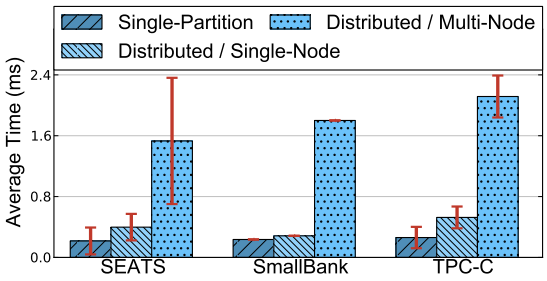
We implemented Hermes in H-Store and measured the system's performance using three OLTP workloads. Our results demonstrate that speculative execution improve the system's overall throughput by 211–771% while reducing latency by 74–88%. We also evaluate an optimistic speculation strategy [138] and show that Hermes outperforms it by up to 113%.

6.1 Distributed Transaction Stalls

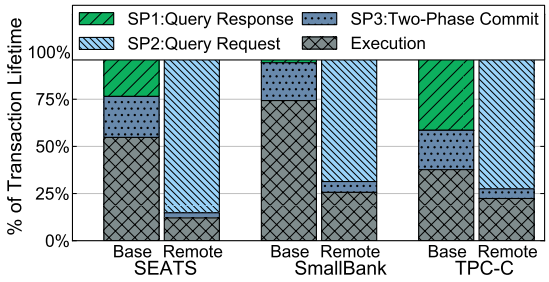
To illustrate how multi-partition transactions are a bottleneck in distributed DBMSs, we now discuss the different types of stalls that can occur because of them.

In H-Store, each transaction has exclusive access to its partitions when it executes [211, 224]. At no point will a single-partition transaction block because of a concurrent transaction nor will it ever need to wait for a network message. Distributed transactions, however, encounter *stall points* where the DBMS halts the transaction until a message arrives that allows its execution to proceed. As shown in the timeline diagram in Fig. 6.2, there are three possible stall points for these transactions: when the DBMS waits for a result from a remote partition after the transaction sends a query request (**SP1**); when the DBMS waits for a new query request from the transaction's base partition (**SP2**); and when the DBMS waits for the transaction's 2PC messages (**SP3**).

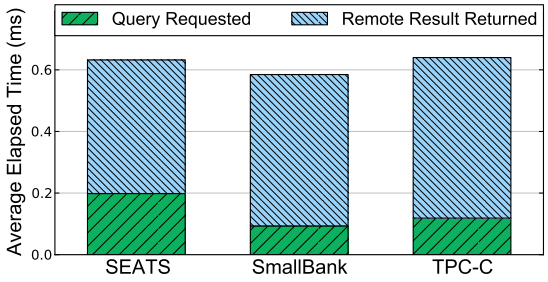
To demonstrate their affect on performance, we ran a series of benchmarks to measure the amount of time that transactions spend at these stall points. For these experiments, we used an instrumented version of H-Store that is able to collect various measurements at runtime [112, 180]. The DBMS begins measuring time at



(a) Transaction Execution Times



(b) Stall Point Times (Distributed / Multi-Node)



(c) Remote Query Execution Times

Figure 6.1: Distributed Transaction Measurements – (a) the average execution time for transactions; (b) the average time spent at stall points for distributed / multi-node transactions; (c) the average elapsed time until a distributed transaction executes a remote partition query and then receives its result.

a transaction's base partition and remote partitions from the moment that the transaction acquires a partitions lock until it commits. We do not include the time the DBMS spends writing recovery log entries or waiting for acknowledgements from replicas, as this overhead is the same for all transactions and is amortizable [128, 155]. For these experiments, we deployed H-Store on a 2-node, 16-partition cluster and execute workloads that are entirely (1) single-partition transactions, (2) single-node, multi-partition transactions, and (3) multi-node, multi-partition transactions. We postpone the details of the execution environment until Section 6.6.

The results in Fig. 6.1a show the total execution times for transactions for the three different workload scenarios. The results in Fig. 6.1b are the percentage of the transaction's lifetime that is spent at the stall points for multi-node, multi-partition transactions. Lastly, Fig. 6.1c shows the amount of time that elapses from when a distributed transaction starts until it (1) executes a remote partition query and (2) receives the result of that query. We now discuss the stall points in the context of these results:

SP1. Waiting for Query Responses

The first stall point is when a transaction blocks at its base partition waiting for the results of a query executed on a remote partition. As shown in Fig. 6.2, when the transaction's control code invokes such queries, the DBMS sends the query request and then stalls until it receives the response [131]. Sending these messages to remote nodes is expensive: multi-node distributed transactions are 3–5× slower than single-node distributed transactions (Fig. 6.1a). The results in Fig. 6.1c show that it takes approximately 0.5 ms for the

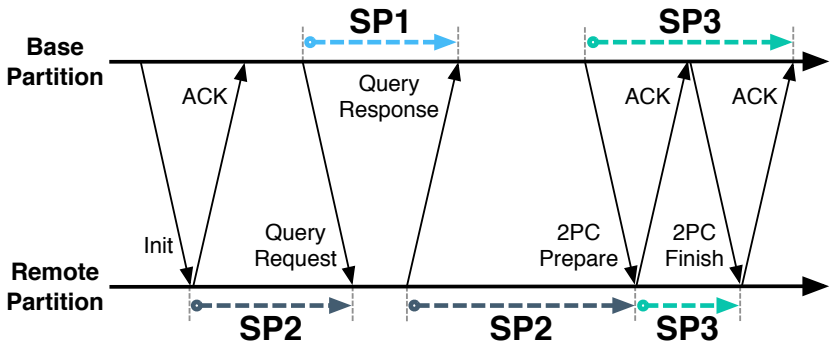


Figure 6.2: Timeline diagram of a distributed transaction with its corresponding stall points. The transaction's control code executes at its *base partition* and invokes queries at the *remote partition*.

DBMS to execute and return the result of a remote partition query. This is roughly twice the amount of time that it takes the DBMS to execute a single-partition transaction.

SP2. Waiting for Query Requests

Once the DBMS grants a remote partition's lock to a transaction, it sends back the acknowledgement to the transaction's base partition. While the transaction holds this lock, no other transaction or query is allowed to execute at that partition or access its data. Thus, this stall point represents the time that the DBMS spends waiting for a query execution request for these remote partitions. Once such a request arrives, the DBMS immediately executes it and sends the results back, but then it stalls again until it receives a new request. Fig. 6.1b shows that the DBMS is idle at this stall point for 68–85% of a distributed transaction's lifetime at its remote partitions. This means that when a distributed transaction holds the lock for a remote partition, it only executes work on that partition transaction for less than 25% of the time that it holds that lock. This stall point is the major reason why distributed transactions degrade the performance of DBMSs that employ a concurrency scheme that is optimized for single-partition transactions [181, 211].

SP3. Waiting for Two-Phase Commit

Lastly, once a distributed transaction finishes, the DBMS sends the 2PC "prepare" message to all of the remote partitions that the transaction locked. The DBMS performs its final checks at these remote partitions to determine whether it is safe to commit that transaction and then send an acknowledgement (or a rejection in case of an abort) back to the base partition. The DBMS cannot begin executing the next transaction at these partitions queue until it receives the 2PC "finish" message that informs it that all of the partitions have agreed to commit the transaction. This is only approximately 2–6% of the time while a transaction holds the lock for its remote partition. But at its base partition, the DBMS has to wait for the second acknowledgement from all the partitions before it is safe to release the lock and execute the next transaction. This accounts for nearly 22% of the transaction's total run time.

These results show the substantial amount of time that the DBMS wastes at these stall points and motivate the need for the DBMS to do something useful instead. Previous work has already shown that speculatively

executing transactions at **SP3** will improve the system's throughput [37, 131, 180, 193]. With this technique, the DBMS executes other transactions at remote partitions before it learns whether the distributed transaction will commit. But as Fig. 6.1b shows, **SP3** only accounts for at most 22% of transactions' total time at their base partition, and just 6% of their total time at remote partitions. This means that existing techniques do not address up to 88% of a transaction's lifetime that the DBMS is idle at the **SP1** and **SP2**.

One approach for dealing with these other stall points is to optimistically execute single-partition transactions whenever a distributed transaction stalls [15, 138]. When the DBMS goes to commit a distributed transaction, it checks whether the speculative transactions violated atomicity and consistency guarantees. If they did, then the DBMS aborts these transactions and schedules them for re-execution. As we show in Section 6.6.3, this approach works reasonably well when there is little skew in the workload. But this is not the case for modern workloads, and thus continually restarting transactions because of conflicts will negate the performance gains of optimistic speculative execution. Furthermore, it does not improve the response time of distributed transactions, since that is limited by the RTT of executing queries on remote nodes.

To improve both the throughput and latency of transactions in a distributed DBMS, we developed the **Hermes** method for scheduling speculative transactions and queries. Hermes is the first work that we are aware of that targets all three distributed transaction stall points. Instead of detecting conflicts *after* they occur, our approach uses machine learning to determine whether speculative work will conflict *before* before it executes. To ensure correctness, Hermes uses a lightweight verification process that only checks that transactions did what they were predicted to do. Thus, when transactions commit, the DBMS knows that the database state is correct.

We now present an overview on how Hermes uses probabilistic models to predict the execution properties of transactions. We then discuss in subsequent sections how Hermes chooses which transactions or queries to speculatively execute at runtime.

6.2 Fast Speculative Execution in Distributed DBMSs

Hermes is a method for scheduling speculative tasks in a distributed DBMS. Such tasks can either be other transactions or queries from the same transaction. The entire process is autonomous and does not require human intervention to maintain once it is deployed.

To identify whether speculatively executing certain tasks will violate transactional guarantees, Hermes must know three things about each transaction before they begin:

1. The queries that it is likely to execute at each partition.
2. Whether it is likely to abort.
3. Its estimated execution time.

One way to acquire this information is for the application to provide hints to the DBMS. But this means additional coding by the developer that would likely require modifications any time that the database or workload changes. Alternatively, the DBMS could collect this information for transactions by simulating their execution [165]. This is insufficient as well because it may fail for cases where a transaction's behavior depends on the state of the database.

```
# PRE-DEFINED SQL STATEMENTS
QueryX = "SELECT * FROM X WHERE X_ID=? AND VAL=?"
QueryY = "SELECT * FROM Y WHERE Y_ID=?"
QueryZ = "UPDATE Z SET VAL=? WHERE Z_ID=?"

# TRANSACTION CONTROL CODE
run(x_id, y_id, value):
    result = executeSQL(QueryX, x_id, value)
    if result == null: abort()
    result = executeSQL(QueryY, y_id)
    # ADDITIONAL PROCESSING...
    executeSQL(QueryZ, result, x_id)
    return
```

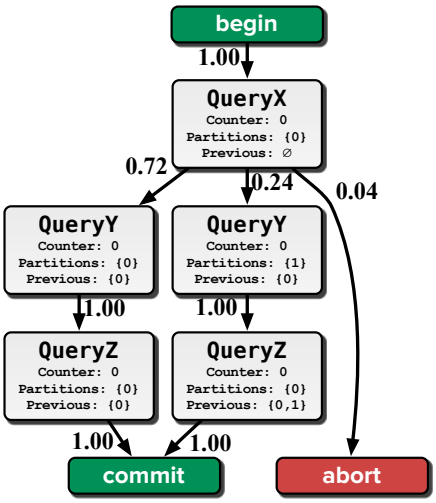


Figure 6.3: An example of a stored procedure's Markov model. Each state in the model represents (1) the query being executed, (2) the partitions that the query will access, (3) the number of times that the transaction has executed this query in the past, and (4) the partitions the transaction has accessed in the past. The `commit`/`abort` states represent the transaction's exit status.

A better approach is to integrate a machine learning framework, like Houdini, in the DBMS that automatically derives this information for each incoming transaction. Such a framework will construct Markov models for each of the application's stored procedures from observing the behavior of previously executed transactions [180]. For each new transaction, Hermes selects a model from Houdini that is tailored for that request and then predicts the path that the transaction will take through that model when it executes. Hermes then derives the three properties listed above for the transaction from this path.

We now present how Hermes uses the runtime transaction estimates (cf. Section 5.3.2) and parameter mappings (cf. Section 5.3.1) from Houdini to identify speculative execution opportunities in a distributed DBMS. We first discuss in Section 6.3 how to speculatively execute other transactions when a distributed transaction stalls. Then in Section 6.4, we describe how to prefetch queries on remote partitions for distributed transactions.

6.3 Speculative Transactions

A *speculative transaction* is one that is executed at a partition without waiting to acquire that partition's lock currently held by a stalled distributed transaction at **SP1**, **SP2**, or **SP3**. This allows the DBMS to execute transactions at that partition while it waits for the distributed transaction to resume. Hermes ensures the database's state is isomorphic to one where transactions execute serially. This means that a speculative transaction cannot read inconsistent data (e.g., reading partial changes from the distributed transaction) and cannot modify the database in a way that causes subsequent transactions to read inconsistent data (e.g., the distributed transaction cannot read its own writes). We define such violations as conflicts.

A transaction *conflicts* with a distributed transaction if speculatively executing it before the distributed transaction commits makes the database inconsistent [48, 93]. Hermes takes a pessimistic approach towards

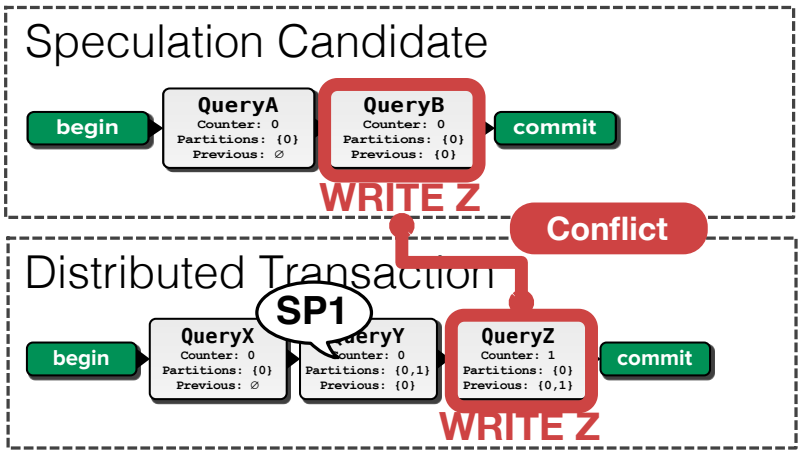


Figure 6.4: An example of comparing the initial path estimate of a speculative transaction candidate with a stalled distributed transaction. The conflict detection rules identified that the `QueryZ` query in the distributed transaction has a write conflict with the `QueryB` in the candidate.

scheduling speculative transactions in that it assumes that such conflicts are common [175]. Thus, it uses pre-computed heuristics to identify whether a transaction might conflict with the stalled distributed transaction before that transaction is executed, as opposed to checking for a conflict after it has occurred [138]. When the DBMS attempts to commit a speculative transaction, as long as that transaction executed the same queries that Hermes originally predicted in its initial path estimate, then the system is guaranteed that a conflict did not occur.

We now discuss the heuristics that Hermes uses to identify non-conflicting candidates. We then describe how the DBMS executes these candidates. We also discuss how this protocol works in the context of the DBMS's recovery mechanism and for deployments with replicated nodes. For this work, we only consider scheduling single-partition transactions for execution; we defer the problem of speculatively executing distributed transactions as future work.

6.3.1 Candidate Identification

When Hermes recognizes that a transaction is stalled at a partition, it examines the queue of transactions waiting to acquire that partition's lock to find the best transaction to speculatively execute at that moment. A queued single-partition transaction is eligible for speculation if (1) the transaction is predicted to finish before the distributed transaction is expected to resume and (2) the transaction is to predicted to not conflict with the distributed transaction.

To determine how much time the DBMS has before the stalled transaction will resume, Hermes uses the time estimates computed in the transactions' initial path estimates. This time is the sum of the state transition time estimates in the initial path. Consider the initial path estimate for a distributed transaction running at partition `0` shown in Fig. 6.4. This transaction has just sent a request over the network to execute `QueryY` at partition `1` and will now stall at its base partition (**SP1**). The run time of a candidate must be less than the estimated stall time to ensure that the distributed transaction can resume executing as soon as it is ready.

For each transaction with a small enough expected run time, Hermes next checks which ones conflict with the distributed transaction. This conflict detection is based on the future execution states of transactions according to their initial path estimates. Some conflicts are detected from static rules (e.g., two queries access different tables, therefore they can never conflict), while others can only be determined at runtime (e.g., two queries access the same table, therefore they only conflict if they use the same input parameters). We discuss how to automatically generate these rules in Section 6.5.

A transaction never conflicts with a distributed transaction that has not executed any queries at a particular partition and is stalled at either **SP1** or **SP2**. The DBMS therefore can execute any single-partition transaction regardless of what data it will access, because the distributed transaction has not observed the state of the database at this partition. A speculative transaction never conflicts with a distributed transaction stalled at **SP3**, since the distributed transaction will never execute more queries. Hermes also skips detection for two procedures that are both read-only or contain queries that access disparate tables. Thus, Hermes only checks for conflicts when the distributed transaction is stalled at **SP1** or **SP2** and has executed at least one query at the partition in question.

Since there may be multiple candidates at a time for a distributed transaction, Hermes must choose which one to execute. We found that selecting the first eligible transaction in the queue provided the best results. More sophisticated approaches, such as choosing the transaction with the shortest expected run time, did not noticeably improve performance and often made it worse because Hermes spent too much time evaluating candidates during the stall points.

6.3.2 Execution

After identifying the best candidate, the DBMS executes that transaction at the target partition as if it had acquired that partition's lock through the normal process. As the transaction executes, Hermes tracks its query invocations and follows the transaction's actual path through the model. The DBMS executes all transactions with an in-memory undo log so that it can rollback in case of an abort.

When the distributed transaction finishes, Hermes decides whether that transaction and any speculative transactions that were executed at the stall points deviated from their initial path estimates in such a way that caused a conflict. A path is considered errant if the transaction reached a state that was not in its initial path estimate. If a speculative transaction executed at **SP3**, or at **SP1** or **SP2** before the distributed transaction executed a query at that partition, then Hermes does not need to perform this final check. Similarly, aborted transactions are returned immediately to the application. The DBMS only checks for conflicts at **SP1** and **SP2** if the distributed transaction executed a query at that partition.

Hermes continues to schedule speculative transactions until either it runs out of non-conflicting transactions in the partition's work queue or the distributed transaction resumes.

6.3.3 Recovery

If a node crashes, the DBMS replays the command log for all of the transactions committed after the last checkpoint to get the database back to the state that it was in before the crash [155]. The DBMS normally re-executes transactions from the log in order according to their original timestamps. But since speculative transactions violate this ordering, additional information is needed to ensure that the recovery matches how

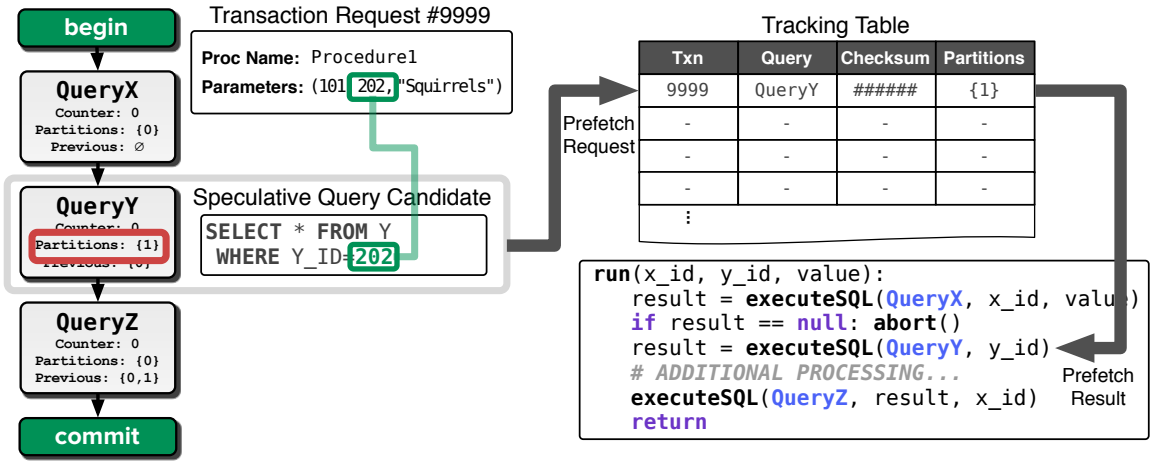


Figure 6.5: Hermes uses a transaction's initial path estimate to identify queries to prefetch. Such queries are dispatched for execution on the remote partition. When the transaction requests the same query in its control code, the DBMS checks the tracking table to see whether the prefetched results have arrived.

transactions were executed at runtime. With Hermes, a transaction's log entry includes all of the speculative transactions that were executed with it and at which stall point the DBMS executed them. This stall point indicates whether the speculative transaction was executed (logically) before or after the distributed transaction finished (i.e., speculative transaction cannot read partial changes from an in-flight distributed transaction). Thus, for each distributed transaction that contains speculative transactions, the DBMS first re-executes all of transactions that were executed at **SP1** and **SP2** before the distributed transaction executed a query. Next, the DBMS replays the distributed transaction, followed by the remaining speculative transactions.

6.3.4 Replicated Environments

The process of executing speculative transactions on partitions' replicas is the same for the distributed transaction's base partition and its remote partitions. At the transaction's base partition, the master replica is responsible for executing the procedure's control code and dispatching queries to other nodes. If a query executes on the transaction's base partition, then the master sends asynchronous query requests to its replicas. This is necessary even if that query is read-only to ensure that the replicas' internal tracking mechanism for tuples' access patterns are synchronized. As Hermes selects speculation candidate transactions on the master replica, it will also send asynchronous messages to its slave replicas. These execution messages are tagged with a sequence number to ensure that the slaves execute in the same order as the master. Only the master replica makes decisions about what to speculatively execute, otherwise the replicas could execute different transactions and get out of sync.

6.4 Speculative Queries

Although executing speculative transactions improves the system's overall throughput, it does not improve the latency of individual distributed transactions. This is because their speed is limited by the network RTT of invoking queries on remote nodes (**SP1**). One could use faster network hardware to reduce this RTT, but

such a change is often not an option for many deployments. For example, if the DBMS is deployed on a cloud computing platform, then the administrator may not have the ability to upgrade the network.

This means that the only way to reduce the amount of time that distributed transactions are stalled at **SP1** is to have query results from remote partitions available at the moment that the transaction needs them. That is, when the transaction's control code dispatches a query for execution at a remote partition, it does not stall because the result is already there [200]. These could be either single-partition queries, or fragments of a multi-partition query (e.g., a COUNT query that spans multiple partitions). In order to do this, the DBMS needs to know what queries the transaction will execute prior to starting so that the system has enough time to prefetch results from remote partitions.

We now present how Hermes enables speculative query execution in a distributed DBMS. The process of identifying speculative query execution opportunities is similar to the process for scheduling speculative transactions. Hermes first uses a transaction's initial path estimate to identify which queries it is likely to execute on remote partitions. The DBMS then asynchronously executes these queries and caches their results at the transaction's base partition. When the transaction invokes a query that was speculatively executed, the DBMS will use the cached result instead of re-executing that query. All of this is transparent to the transaction's control code. At commit time, Hermes verifies that its decisions were correct based on whether the transaction's actual execution path through the target Markov model deviated from its initial path.

Speculative query execution does not require changes to an application's stored procedures. Likewise, no change is needed for the DBMS's recovery mechanism or the execution process in replicated environments, since the database's state is the same regardless of whether transactions execute with speculative queries or not.

6.4.1 Candidate Identification

Just as with speculative transactions, the process of identifying speculative query candidates has both an off-line, static analysis component and a runtime component.

Prior to deployment, Hermes analyzes the queries in the application's stored procedures to determine which ones the DBMS can speculatively execute. A query is eligible if the DBMS can extract all of the values for that query's input parameters from its procedure's input parameters. That is, if the values for a query's parameters are derived from the values sent by the application, as opposed to coming from the output of a previously executed query [187]. Hermes identifies the queries with this property using the parameter mappings that it generates from a sample workload trace. As shown in the example in Fig. 6.5, Hermes uses a mapping to fill in all of the input parameters for `QueryX` and `QueryY` from the procedure's parameters, but not for `QueryZ`. Thus, `QueryZ` is not eligible for speculation since it is incomplete.

Next, for each procedure with at least one query that is eligible for speculative execution, Hermes analyzes that procedure to find pairs of potentially conflicting (i.e., non-commutative) queries. For example, two read-only queries never conflict because their results are the same regardless of their execution order. But if the first query reads a table and then the second one modifies that same table, then their execution order does matter. The DBMS will verify that the execution order of such queries is correct. Hermes detects these conflicts using the rules described in Section 6.5.

At runtime, when a new transaction arrives, Hermes uses that transaction's initial path estimate to find

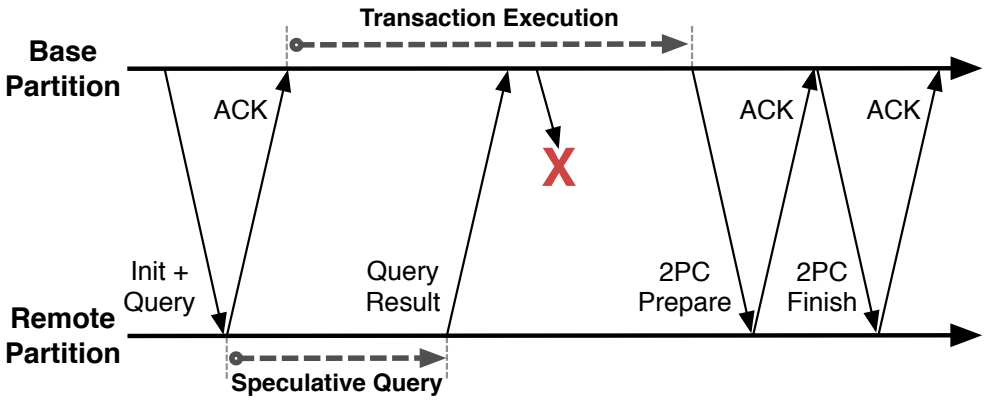


Figure 6.6: Timeline diagram of speculatively executing a query for a distributed transaction. The query result arrives at the transaction's base partition before it is needed, thus the DBMS will not send out a request when the procedure control code invokes it.

execution states that correspond to eligible remote partition queries. Hermes evaluates the positions of the candidate states in the initial path to determine whether executing them out-of-order will create a conflict with other queries that the transaction is expected to execute at their target partitions. For example, consider a transaction that updates a table and then immediately read from it. If the read query is eligible for speculation but the update is not, then the read query is not a speculation candidate since it must occur after the update. Unlike with speculative transactions, where Hermes only selects one candidate at a time, all of the query candidates are selected together. If no such conflict exists, then Hermes marks a query as a speculative execution candidate for the transaction.

6.4.2 Execution

For a transaction with speculative query candidates, the DBMS must send the query invocation requests to the remote partitions before that transaction starts running. As shown in Fig. 6.6, the DBMS piggybacks these requests onto the partition lock acquisition messages. When this message arrives at a node, the DBMS holds the query requests until the transaction acquires the lock for that partition. After the transaction is granted the lock, the speculative queries for that partition are released for execution. Once they complete, their results are sent back to the transaction's base partition. The DBMS does not wait for the transaction to acquire the locks that it needs at a node before executing queries at a remote partition. It is therefore possible that the DBMS could receive results for speculative queries before the transaction's control code starts running. Hence, the DBMS must accommodate prefetch results that arrive both before and after the transaction has started.

To facilitate this, Hermes maintains a *tracking table* in the DBMS for transaction's speculative queries. This enables Hermes to match runtime query invocations with the results from speculative queries, thereby preventing the DBMS from executing the same query twice. It is integrated with the DBMS's internal infrastructure that coordinates multi-partition query execution. When a transaction invokes a regular query (i.e., non-speculative), Hermes checks its tracking table to determine whether that particular invocation was requested as a speculative query based on its (1) input parameters, (2) relative execution order in the transaction, and (3) accessed partitions. If it was, then Hermes links the DBMS's internal record for the regular

query to the speculative query's tracking table entry.

When a transaction commits, Hermes verifies all of the transaction's speculative queries executed (1) with the correct input parameters and at the correct partitions, and (2) in the correct order relative to non-commutative queries.

To determine whether queries executed correctly, Hermes checks its tracking table for entries that are not linked to a regular query invocation. Such queries are deemed *unnecessary* because the transaction ended up not needing them. For each partition that executed unnecessary queries, Hermes must decide if it is still safe to commit the transaction, or whether it needs to revert the errant queries' changes. If all of the speculative queries at a partition were read-only, then the transaction can commit since the superfluous queries are discarded. But if the speculative queries modified the partition, then Hermes checks whether those changes were accessed by the same transaction or a different (speculative) transaction. If they were read by the distributed transaction, then that transaction is restarted because the changes may have caused other modifications. If the changes were read by a speculative transaction, then the speculative query is reverted and that transaction is restarted. The distributed transaction still commits at other partitions, since the transaction did not read these modifications.

A speculative query is deemed *out-of-order* if the DBMS executes it in a different order relative to conflicting queries in the same transaction at a partition. To determine whether a query is out-of-order, the DBMS includes an ordered list of the query invocations for the transaction derived from its actual path execution in the 2PC messages. Hermes checks its tracking table to see whether the DBMS executed the queries in the correct order. This is necessary because there is a race condition between when the speculative query is executed and when the first non-speculative query arrives at that partition. If any speculative query was executed in the incorrect order relative to a non-commutative query, then the distributed transaction is restarted.

Hermes monitors how often transactions abort due to these above conditions. If the percentage of successful transactions reaches below a threshold for a particular procedure, then it will disable all speculative query execution for that procedure. This avoids the problem of the DBMS continually aborting transactions because of conflicts.

6.5 Conflict Detection

Hermes' conflict detection rules are the key component that allows it to safely schedule speculative tasks in Sections 6.3 and 6.4. These rules are generated off-line from analyzing the queries in the application's predefined stored procedures. Conflicts are detected based on pairs of queries: two queries are said to conflict if executing them in reverse order (as part of either a speculative transaction or a speculative query) would violate isolation guarantees [33, 224].

There is an inherent trade-off between the amount of time that Hermes spends detecting conflicts versus the time that the DBMS spends executing speculative operations. In the case of speculative transactions, a thorough detection algorithm may allow Hermes to find more candidates at a particular stall point, but it will take longer to discover them. Likewise, for speculative queries, a faster algorithm may incorrectly declare that out-of-order operations conflict, causing the DBMS to restart the transaction. A slower algorithm can avoid these unnecessary restarts but will slow down the total execution time of transactions.

Given this, we present two algorithms for detecting conflicts at runtime. Our first approach is a coarse-grained detection that compares queries based on what tables they access. The second algorithm analyzes the input parameters and predicates of the queries to estimate whether they access the same rows. It is important to note that both approaches never have false negatives, but they may have false positives. We compare the efficacy and performance of these two detection algorithms in Section 6.6.2.

6.5.1 Table-level Conflict Detection

The first type of detection is based on whether two queries access the same tables in a conflicting manner. Given two queries q_1 and q_2, Hermes identifies that q_1 conflicts with q_2 if any of the following conditions are true:

1. Both q_1 and q_2 are either an INSERT, UPDATE, or DELETE on the same table.
2. q_1 is an INSERT or DELETE on a table, and q_2 is a SELECT on that same table.
3. q_1 is an UPDATE that modifies a table, and q_2 is a SELECT references the same columns modified in q_1 in either its output or WHERE clause.

The intuition behind the first rule is that the DBMS cannot allow q_1 to modify a table that is also modified by q_2, since the second query may violate the table's integrity constraints when it otherwise would not if the queries were executed in the proper serializable schedule order. The other two rules ensure that modifications made by q_1 are not read by q_2. For example, if q_1 is executed in a speculative transaction and q_2 is executed later by the distributed transaction after it unstalls, then these two queries conflict because the distributed transaction should not be able to read any modifications made by the speculative transaction.

6.5.2 Row-level Conflict Detection

There are cases where table-level detection generates false positives for queries that obviously do not conflict. For example, if q_1 updates a table and q_2 reads that same table, a conflict only arises if these two queries access the same row(s). Thus, Hermes' row-level conflict detection seeks to overcome this limitation and only report true conflicts. But since these rules are evaluated at runtime *before* the queries are executed, Hermes needs a way to identify what rows each query accesses without having to run it.

Normally predicting rows an arbitrary query accesses is non-trivial. But we can rely on the observation that most transactions in OLTP applications access a small number of records from a single table using indexes [211]. This means that for the queries with this property, Hermes can derive what rows those queries access based on the values their transaction input parameters using the same parameter mapping technique used to schedule speculative queries (cf. Section 6.4.1). For example, if a SELECT query uses a table's primary key index, then Hermes can use the mappings to identify which of the transaction's input parameters correspond to the columns in the index from the query's WHERE clause. A query will access a single row if its WHERE clause contains equality predicates on unique indexes without a disjunctive operator (i.e., OR). Such queries are common in OLTP applications [49, 180]. These rules only compare the rows that queries potentially access; it does not matter whether those records actually exist.

Given two queries q_1 and q_2, Hermes identifies that q_1 conflicts with q_2 if any of the following conditions are true:

1. Both q_1 and q_2 are INSERTs on the same table and key.
2. q_1 is an INSERT, UPDATE, or DELETE on a table, and q_2 is a SELECT that accesses that same table using a join, range scan, or aggregate.
3. q_1 is an INSERT or DELETE on a table using a key, and q_2 is a SELECT on the same table and key.
4. q_1 is an UPDATE that modifies a table using a key, and q_2 is a SELECT references the same columns modified in q_1 in either its output or WHERE clause.

For the first rule, we disallow queries that insert rows with the same key. The second rule is broadly defined because q_2 is likely to access multiple rows, thus Hermes no way of knowing whether that query will access the same record accessed in q_1. The remaining two rules are the same as the last two in the table-level rules.

6.6 Experimental Analysis

To evaluate Hermes' speculative execution techniques, we integrated it with H-Store [3] and ran several experiments using three OLTP benchmarks with differing workload complexities: SEATS, SmallBank, and TPC-C (cf. Appendix A). For each benchmark, we trained Hermes' models and parameter mappings using a workload trace of 100,000 transactions collected over a 30 minute period.

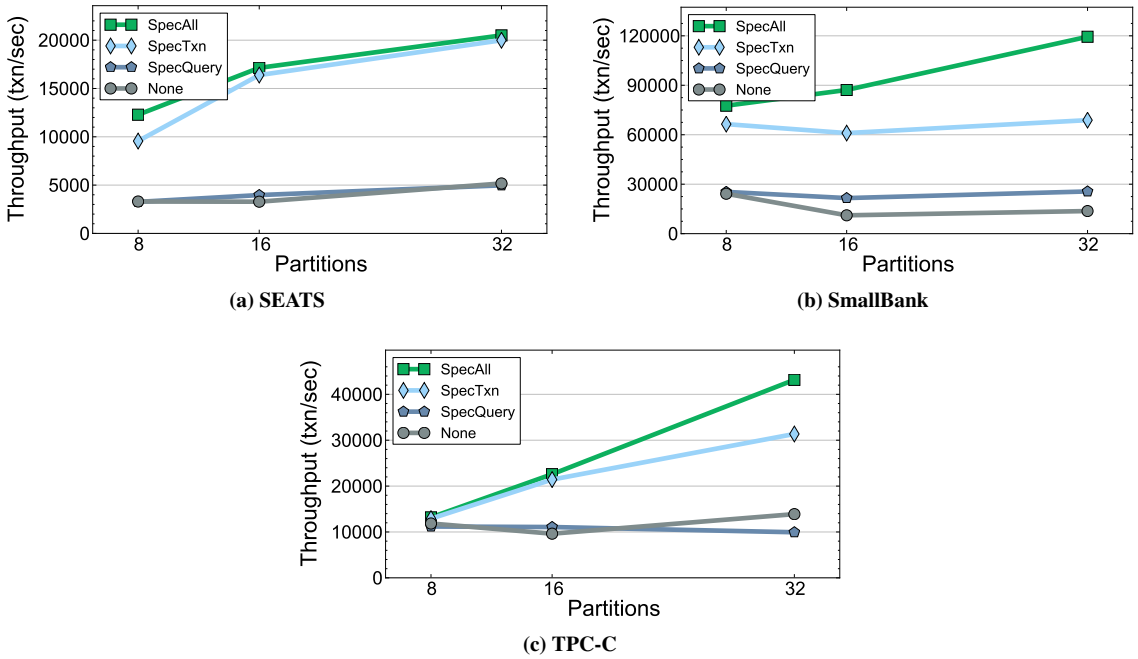
All of the experiments were conducted on a cluster at MIT. Each node has a Intel Xeon E7-4830 CPU running 64-bit Ubuntu Linux 12.04 with OpenJDK 1.7. We use the latest version of H-Store with command logging enabled to write out transaction commit records to a single 7200 RPM disk drive. The nodes are in a single rack connected by a 10GB switch with an average RTT of 0.42 ms.

6.6.1 Performance Evaluation

We first compare H-Store's performance when using Hermes' speculative execution techniques. For each benchmark, we ran H-Store with the following system configurations and measure the throughput and latency of transactions:

SpecTxn: Speculative Transactions (Section 6.3)
SpecQuery: Speculative Queries (Section 6.4)
SpecAll: Speculative Transactions + Queries
None: No Speculation

We use three cluster sizes in this evaluation (1, 2, and 4 nodes) with eight partitions per node. The DBMS's execution engine threads are given exclusive access to a single core to improve cache locality. Remaining cores are used for the networking and administrative functionalities of H-Store. Transaction requests are submitted from up to 800 clients running on a separate node in the same cluster. Each client submits transactions to any DBMS node in a closed loop (i.e., it blocks after it submits a request until the



(a) SEATS

(b) SmallBank

(c) TPC-C

Figure 6.7: Throughput Measurements

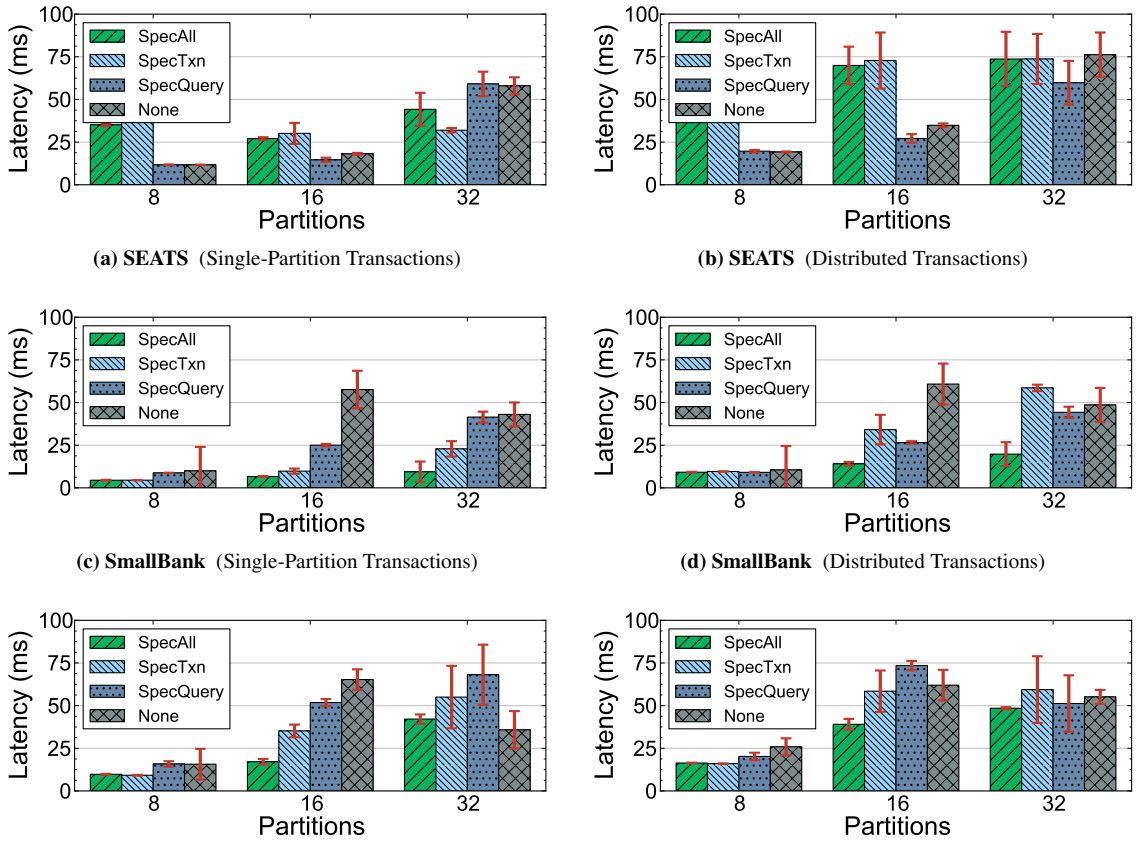
result is returned). For the configurations with speculative transactions, the clients submit additional single-partition transaction requests to ensure that the DBMS's queues always contain speculation candidates. The overall number of distributed transactions executed is the same across all configurations per cluster size. We also configure Hermes to use row-level conflict detection for these experiments (cf. Section 6.5.1).

For each cluster size and configuration combination, we execute each benchmark three times and report the average results. In each trial, the DBMS "warms-up" for 60 seconds and then the performance metrics are collected for five minutes. The throughput results in Fig. 6.7 are the number of transactions completed divided by the total time (excluding the warm-up period). The latency results in Fig. 6.8 are measured as the from when the client submits a request to when it gets the transaction's result. We report the single-partition and distributed transaction latencies separately.

We now discuss several aspects of the results from these experiments. We first note that for all cluster sizes and workloads, the None configuration has both lower throughput and longer latencies. This validates previous results that demonstrated that distributed transactions hinder shared-nothing DBMSs from scaling [131, 180].

With the SEATS benchmark, we see in Fig. 6.7a that SpecAll and SpecTxn improve the DBMS's throughput by roughly the same amount, whereas there is no improvement for SpecQuery. This is because there are few remote queries to prefetch in this workload.

The results for SmallBank highlight the individual benefits of the SpecTxn and SpecQuery techniques. In Fig. 6.7b, we see that SpecTxn improves H-Store's throughput by 2× over the None configuration, but that with SpecQuery it only improves slightly. This is expected, since with SpecTxn the DBMS is speculatively



(a) SEATS (Single-Partition Transactions)

(b) SEATS (Distributed Transactions)

(c) SmallBank (Single-Partition Transactions)

(d) SmallBank (Distributed Transactions)

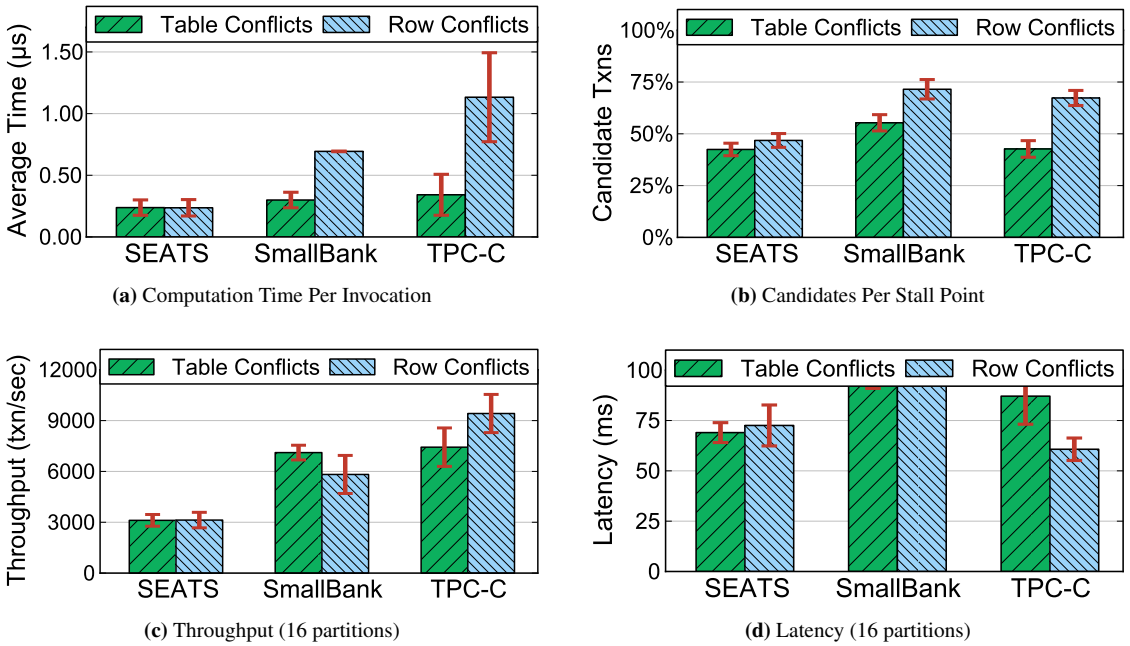
(e) TPC-C (Single-Partition Transactions)

(f) TPC-C (Distributed Transactions)

Figure 6.8: Latency Measurements

executing more single-partition transactions. Likewise, Fig. 6.8c shows that SpecTxn and SpecQuery have lower single-partition transaction latencies because they spend less time waiting in the partitions' queues. But in Fig. 6.8d, SpecQuery has lower distributed transaction latencies than with SpecTxn. This is because Hermes prefetches all remote partition queries for every distributed transaction in the workload. It is the overhead of distributed transactions that prevents H-Store's performance from scaling with SpecTxn in Fig. 6.7b. Thus, when using speculative transactions and queries together in SpecAll, the DBMS's throughput improves by 770.8% and its latencies are reduced by 88.4% for the 32-partition cluster.

Lastly, TPC-C contains a transaction (`NewOrder`) that invokes a remote partition query that uses the output of a previous query as its input. This query cannot be prefetched, and thus these transactions will always stall at **SP1**. The results in Fig. 6.7c show that DBMS's throughput with SpecQuery does not improve because of this. Even though the DBMS is unable to prefetch this one query, the ones that it can prefetch do provide some benefit. For example, H-Store almost achieves the same throughput for the SpecAll and SpecTxn configurations on the 8- and 16-partition clusters, but the performance with SpecAll is much better on the 32-partition cluster. This is because distributed transactions are more likely to need data that is not on the same node as their base partition. When all of a transaction's partitions are local, the benefit of speculative



(a) Computation Time Per Invocation

(b) Candidates Per Stall Point

(c) Throughput (16 partitions)

(d) Latency (16 partitions)

Figure 6.9: Conflict Detection – (a) the average length of the computation time per invocation; (b) the percentage of queued transactions correctly identified as eligible for speculative execution; (c) the DBMS's throughput for each detection algorithm; (d) the transactions' latencies for each detection algorithm.

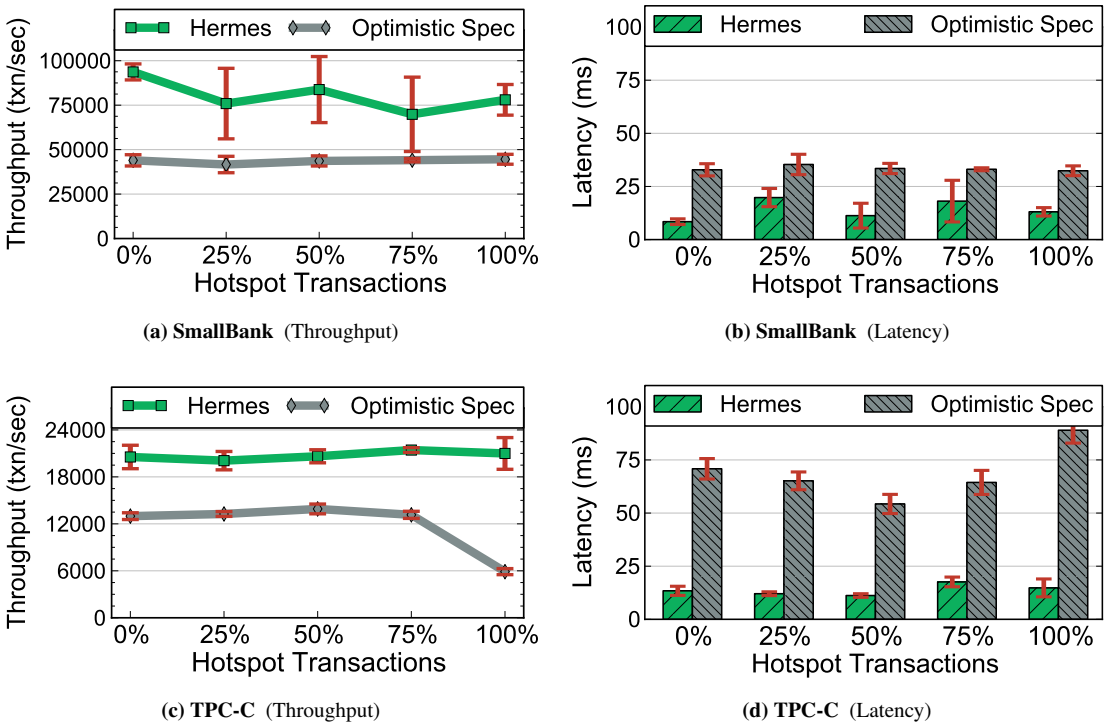
queries is reduced; this why the transaction latencies in Fig. 6.8f for SpecQuery are roughly the same as for None on the 8-partition cluster. Hence, on the 32-partition cluster the advantages of prefetching are more pronounced because of the network latency.

In general, these results show that by speculative transactions and speculative queries by themselves do not scale H-Store. But when combined together, the DBMS achieves both higher throughput and lower latencies on larger cluster sizes.

6.6.2 Conflict Detection Comparison

Next, we investigate the performance characteristics of Hermes' two conflict detection algorithms (cf. Section 6.5) when identifying candidate transactions for speculative execution. For each algorithm, we measured its (1) computation time and (2) candidate identification yield. The former is the amount of time that Hermes spends looking for candidates in a partitions' lock queues. The candidate yield is the percentage of transactions that an algorithm identifies as non-conflicting at runtime. For these experiments, Hermes will examine every transaction in a partition's queue and counts the number of candidates it identifies (as opposed to stopping as soon as it finds one). We ran all the three benchmarks again in H-Store using each detection algorithm and collected profiling information whenever the DBMS requests Hermes to find a candidate transaction. All of these experiments were conducted on a 2-node, 16-partition cluster using the SpecTxn configuration.

The results in Fig. 6.9a show that the table-level detection algorithm is 131.8% and 231.4% faster than



(a) SmallBank (Throughput)

(b) SmallBank (Latency)

(c) TPC-C (Throughput)

(d) TPC-C (Latency)

Figure 6.10: OCC Comparison – Performance measurements for H-Store using Hermes' pessimistic scheduling or optimistic scheduling. The "hotspot transactions" are the percentage of the the workload that all target a fixed subset of the database.

row-level detection for SmallBank and TPC-C, respectively. This is again because that algorithm only compares queries based on the tables that they access. The results in Fig. 6.9b show that the additional time used in the row-level detection enables Hermes to identify 29.3% more candidates for SmallBank and 57.6% more candidates for TPC-C. There was no measurable difference between the two algorithms computation time for the SEATS workload, but the row-level detection yielded 10.3% more candidates.

One interesting thing to note is that the standard deviations are greater for the TPC-C measurements in Fig. 6.9a than for the other two workloads. This is because its distributed transactions stall at **SP1** and **SP2** multiple times and thus the number of queries in the transaction path estimates that the detection algorithms must analyze varies. TPC-C's transactions also execute more queries on average than the transactions in the other two workloads, which means that it takes longer for Hermes to evaluate them each time.

The results in Figs. 6.9c and 6.9d show that there is little difference in the runtime performance of the DBMS using either detection algorithm. The row-level detection improved throughput only slightly over the table-level detection algorithm. It is likely that the advantages of row-level detection will be more apparent with more complex workloads, but based on our experience we found that the workloads that we used here are representative of a larger number of real-world OLTP applications.

6.6.3 Optimistic Scheduling Comparison

Finally, we compare Hermes to a scheduling approach based on optimistic concurrency control (OCC) [15, 138]. With OCC scheduling, when a distributed transaction stalls at a partition, the DBMS executes the next single-partition transaction in that partition's lock queue without first checking whether it conflicts. The DBMS tracks the read/write sets at each partition for all transactions as they execute. At the prepare phase in 2PC, the DBMS checks whether any of these speculative transactions conflict with the distributed transaction based on their read/write sets. If there was a conflict, then both the distributed transaction and the speculative transaction are aborted and restarted. These aborts cascade to any speculative transaction that read data modified by another aborted transaction.

For this evaluation, we use the SmallBank and TPC-C workloads on a 2-node, 16-partition cluster. A portion of the database in each benchmark is declared as the hot spot and we vary the percentage of the transactions that access only those records [45]. The number of distributed transactions is fixed at 10% of the original workload. We also modified the benchmark's workload generators so that all distributed transactions access data located on two nodes. This increases the length of stall points, thereby maximizing the time that the DBMS has to execute speculative transactions. We deployed Hermes in H-Store using the SpecTxn configuration, since the OCC scheme is unable to prefetch remote partition queries.

The results in Figs. 6.10b and 6.10d show that the transactions in H-Store with OCC have on average 74.3–81.5% longer latencies than with Hermes. The measurements in Figs. 6.10a and 6.10c show that the OCC has a 1.8× lower throughput. This is due to the nature of H-Store's architecture [133]. In H-Store, transactions' tasks at a partition are processed by a single-threaded execution engine that has exclusive access to the data at that partition. With OCC, it takes longer for the DBMS to complete 2PC because it needs to check the read/write sets for all of the speculative transactions at each partition. Thus, the DBMS cannot execute any other tasks at the partition during this time.

The throughput results for TPC-C in Fig. 6.10c are notable because they demonstrate the known problem in OCC for skewed workloads. When the entire workload targets the hotspot region in the database, the likelihood that a speculative transaction will conflict with the stalled distributed transaction is great. Thus, with OCC, H-Store has to abort more transactions, thereby causing the throughput to drop. Hermes avoids this problem because it only schedules transactions that it knows will not conflict. Our results in Fig. 6.10a indicate that this is not a problem for SmallBank. This is because some of its transactions only read data from the hotspot, whereas in TPC-C the transactions almost always modify the database.

6.7 Conclusion

Our Hermes approach uses Markov models to forecast the behavior of transactions in OLTP workloads and then enables the DBMS to (1) interleave transactions whenever the distributed transaction stalls and (2) prefetch remote partition queries before a transaction starts. At runtime, Hermes chooses operations to speculatively execute in the DBMS if they will not conflict with the stalled distributed transaction. To evaluate our work, we integrated Hermes into the H-Store distributed OLTP system. The results from our experiments show that our approach improves the DBMS's throughput by an average of 467.9% and reduces transactions' latencies by 108.6% on average.

Chapter 7

Related Work

Since transaction processing was the first workload targeted by the original database management systems, there is a long history and an extensive corpus on the problems on executing transactions correctly and efficiently in these systems. Thus, we first provide a review of the previous academic and industrial systems for transaction processing that are similar to H-Store. For this discussion, we will examine the various components of these systems as they relate to H-Store, including concurrency control schemes, recovery mechanisms, fault tolerance measures, and query processing. We then discuss the previous work in automatic database design, predictive transaction modeling, and speculative execution techniques.

7.1 Database Management Systems

The advantages of scaling a database across multiple nodes were apparent during the initial research of relational DBMSs in the 1970s. But all of these earlier distributed DBMSs focused on disk-based systems and how to complex joins efficiently [61, 74]. At the same time, others began exploring main memory-oriented DBMSs [72].

We now discuss the history of the development of these two types of systems, as well as those systems that are both distributed and main memory-oriented. We restrict this review to focus only front-end transaction processing systems. There is a prodigious amount of previous work on distributed DBMSs for other types of problem domains, including large-scale data analytics [71], scientific databases [65], and stream processing [14] that are beyond the scope of this dissertation. Although these other workloads are different than what H-Store supports, many of the core concepts for these systems DBMSs (e.g., concurrency control, high availability, fault tolerance) are still the same.

7.1.1 Distributed Systems

Many of the fundamentals of distributed database management systems came from the seminal work in the SDD-1 project. SDD-1 was the first DBMS specifically designed for distributed operation across a cluster of shared-nothing nodes [191]. It used a centralized coordinator to manage distributed query processing [34], concurrency control [27, 33], and system availability [110]. SDD-1 split databases into disjoint fragments stored on disk using horizontal and vertical partitioning. It is also noteworthy for supporting serializable

transactions using timestamp-based ordering and conflict graph analysis instead of the traditional two-phase locking that was used in other early DBMSs [28]. SDD-1's transaction coordinator could detect "classes" of transactions that would never conflict and thus could be executed with little or no synchronization overhead [191]. This is similar to our predictive modeling framework in H-Store for identifying transactions that can execute with minimal concurrency control protection.

After SDD-1, the teams behind the two pioneering, single-node DBMSs, System R [24] and INGRES [209], both created distributed versions of their respective systems. IBM's R* was also a shared-nothing, disk-oriented distributed DBMS like SDD-1 [225]. The main difference, however, was that the coordination of transactions in R* was completely decentralized just line in H-Store. R* used distributed two-phase locking protocol where transactions locked data items that they needed to access directly at nodes [160]. Since deadlocks are inevitable in such a system, each node would independently run a distributed deadlock detection algorithm that polled other nodes to identify cycles in transactions' dependency graphs [171]. The R* researchers explored various optimizations to two-phase commit in [160] that are relevant to H-Store, such as *presumed commit* (cf. Section 3.6.4). Another interesting feature of R* was its ability to relax consistency guarantees to allow read-only snapshots of previous states of tables on multiple nodes that are periodically refreshed [154, 225].

The distributed version of INGRES also used decentralized two-phase locking with centralized deadlock detection [204, 205]. The system is mostly remembered for its dynamic query optimization algorithm that recursively breaks a distributed query into smaller pieces [81].

Tandem's Encompass was the first commercial distributed relational DBMS [196], but it did not support SQL or data independence [42]. In 1986, Tandem released the NonStop SQL system that was built on top of the earlier Encompass architecture [212]. The initial version only supported single-node transactions using strict two-phase locking; support for multi-node transactions were added in 1989. The system was noted for being one of the few DBMSs at the time that were able to scale its performance linearly as new nodes were added to the system [70].

Around the same time that NonStop SQL was being developed, researchers at the University of Wisconsin–Madison developed the Gamma system [73]. This was another shared-nothing DBMS that supported multiple horizontal partitioning strategies [70], including round-robin, range, hash, and a hybrid approach [89]. Gamma was similar to distributed INGRES in that it used two-phase locking with a combination of local and global deadlock detection [75].

In the Bubba system [40], developers write transactional control code and queries using a special-purpose programming language (instead of SQL). When this code is compiled, the Bubba compiler detects operations that can execute in parallel based on the database's design specification. This is similar to our approach in H-Store for compiling SQL statements into query plans before the system starts and then computing conflict detection rules based on those plans (cf. Section 6.5).

There are also several other distributed DBMS research projects that were started after H-Store. In 2010, the RelationalCloud [67] project from MIT was built using the same distributed transaction coordinator framework from an earlier version of H-Store [130, 131]. The main differences between RelationalCloud and H-Store is that it was more akin to a middleware system, whereas H-Store is a monolithic (i.e., tightly

coupled) system. The RelationalCloud uses a centralized router [213] that dispatches queries over shared instances of MySQL, thus it was inherently disk-based. The Granola system used the same lightweight serial concurrency control model from H-Store for efficient transaction execution [64].

7.1.2 Main Memory Systems

In the 1980s, researchers at the University of Wisconsin–Madison explored the design issues related to main memory DBMSs [72]. From this work came that the primary goals of a main memory DBMS were to reduce the overall computation time and to use as little memory as possible. The seminal research by Lehman and Carey provide the foundation for many aspects of main memory DBMSs, including indexes [146], query processing [147], and recovery algorithms [148]. This work was conducted as part of Lehman's dissertation [145]. Garcia-Molina and Salem also provide a good summary of the differences between traditional, disk-oriented DBMSs and main memory DBMSs [88].

Dalí [38, 125] was the first main memory DBMS that was designed for high-performance transaction processing. It uses a memory mapping-based architecture, where the database is stored in virtual memory address space and queries access data through pointers. The runtime components, such as the concurrency control manager and logging services, communicate via shared memory. Dalí was later commercialized as DataBlitz [26]. H-Store does not use Dalí's memory mapping-based approach and instead manages a databases storage in memory itself. This distinction is important because of H-Store's single-threaded execution model; with virtual memory, if a transaction attempts to access a page not in memory, then the execution engine will stall.

Two other commercially available main-memory DBMSs are Oracle's TimesTen [139, 219] (originally SmallBase [118] from Hewlett-Packard Laboratories) and IBM's solidDB [152]. Both are primarily single-node systems that can replicate a database across multiple nodes. TimesTen uses single-version locking with multiple granularities (e.g., row, table, and database) [139]. solidDB supports both main memory-resident and disk-resident tables; for the former, the system provides single-version locking at either table- or row-level, while for the latter it supports both optimistic and pessimistic concurrency control [152].

The architecture of the New York University's K system proposed in [224] is probably the most similar to H-Store's design. Their main memory execution engine processes transactions sequentially at in-memory partitions that are each assigned to a single CPU core. The system also uses a logical logging scheme similar to the one that we describe in Section 3.7. They also provide a system for developers to manually specify at runtime which types of transactions conflict. The key difference, however, is that K did not support distributed deployments or multi-partition transactions.

The Shore-MT project was the leading research rival for H-Store during this author's time in graduate school [127]. They developed a single-node, shared-everything main memory DBMS with a multi-threaded kernel [126, 129] that is based on the Shore DBMS [47]. DORA is an OLTP execution engine built on top of Shore-MT [178]. Instead of assigning transactions to threads, as in traditional DBMS architectures, DORA assigns partitions to threads. That means when a transaction needs to access data at a specific partition, the transaction handle must be sent to the thread that manages that partition and then wait to be executed by it. This similar to H-Store's partitioning model, except that DORA supports multiple record-level locks per partition (instead of one lock per partition) [177].

Hekaton [76] is a main memory table extension for Microsoft's SQL Server. The administrator designates certain tables in their application as special in-memory tables that are managed by Hekaton. The DBMS supports transactions that access the in-memory tables together with regular, disk-resident tables. Instead of the partitioned model employed in H-Store, Hekaton uses multi-version concurrency control scheme with lock-free data structures [116, 143]. They also compile queries that only main memory tables into machine code for faster execution, which was a technique applied earlier in System R [52].

The HyPer DBMS [134] is a hybrid main memory system that is designed to execute both OLTP and OLAP queries simultaneously. Similar to H-Store, OLTP queries in HyPer are executed serially at partitions without the need for a heavyweight concurrency control scheme. But to prevent longer running OLAP queries from blocking regular transactions for too long, HyPer creates in-memory snapshots by forking the DBMS's process and executes those queries on separate CPUSs in the forked process. These snapshots are then periodically refreshed, similar to the approach used in R* [160]. Like Dalí, HyPer relies on virtual memory to support databases that are larger than the amount of available memory. We discuss in Section 8.7 our future plans to extend H-Store to support mixed OLTP/OLAP workloads like HyPer.

7.1.3 Distributed, Main Memory Systems

There are a number of distributed, main memory DBMSs that have been proposed, each with different design decisions.

One of the earliest of these systems is PRISMA/DB [135]. PRISMA uses two-phase locking concurrency control and 2PC to execute general purpose workloads over a partitioned database using main memory storage [21]. PRISMA differs from H-Store in that it has a centralized transaction coordinator and supports dynamic runtime compilation of query operators into machine code.

Another system that was developed around the time of PRISMA was the ClustRa Telecom Database [123]. Instead of partitioning a database across multiple nodes, ClustRa fully replicated the database at each node.

ScyPer [163] is a distributed version of the HyPer DBMS that supports simultaneous OLTP and OLAP workloads. The database is stored on two separate sets of nodes, where the first set contains only a primary node where all transactions are executed and secondary set contains a potentially stale copy of the database. The primary node streams updates to the secondary nodes. All long-running OLAP queries are only executed on the secondary nodes.

SAP's HANA [144, 199] is another distributed, main memory hybrid DBMS. Like ScyPer, this system supports simultaneous OLTP and OLAP workloads running in the same system. But HANA splits a database into separate in-memory row and columnar stores [84], whereas ScyPer's snapshots are byte-for-byte copies of the in-memory row store. Each storage type in HANA can be split across multiple machines using either horizontal or vertical partitioning. An application's OLTP transactions operate on the row-storage data while the OLAP queries use the column-storage [13]. HANA's OLTP execution engine is based on the P*TIME [51] DBMS. It uses optimistic concurrency control [31, 138] to avoid runtime contention for locks in shared data structures.

Two notable research systems are Stanford's RAMCloud and Yale's Calvin projects. RAMCloud [174] is a scalable main memory resident key-value store that provides low latency operations (i.e., less than 5 μs per operation) on cloud computing platforms. It is not a general purpose DBMS like H-Store, since it

does not provide certain features, such as secondary indexes and multi-object transactions. Calvin [218] is a transaction scheduler middleware that batches together commutative operations for multiple transactions by exploiting knowledge about their future read/write requirements. It sends batches to separate DBMS nodes where they are executed without cross communication. As we discuss in Section 7.4, Calvin does not support stored procedures and lacks any mechanism to determine transaction's dependency information automatically.

We next discuss the leading systems that are part of the NoSQL and NewSQL movements.

7.1.4 NoSQL Systems

During the beginning of the 2000s, developers turned to *NoSQL* systems to build large-scale, Internet-based applications [49]. These NoSQL systems are mostly characterized as forgoing the relational data model and traditional features found in single-node DBMSs in favor of availability and scalability.[1] That is, they favor availability and network partition-tolerance over consistency [222]. Notable proprietary systems include Google's BigTable [54], Yahoo!'s PNUTS [60], and Amazon's Dynamo [69]. Later in the decade, others began re-implementing these ideas in open-source projects, such as HBase [2] (i.e., BigTable), Cassandra [140] (i.e., Dynamo). A non-distributed NoSQL DBMS, called Redis [8], is a non-transactional, single-threaded execution engine that is similar to the one used in H-Store (cf. Section 3.2.2).

The first generation NoSQL systems typically only allowed applications to perform consistent operations over a single table key space. This means that operations, such as joins, must be performed in the application. By reducing the functionality of the DBMS, it is easier to distribute operations and data across multiple machines than general purpose DBMS [222]. Newer NoSQL systems provide support for multi-key and multi-node transactions [82].

7.1.5 NewSQL Systems

After the general purpose version of H-Store was created in 2008 [133], the code was forked and commercialized as VoltDB [162]. Since then, several other high-performance transaction processing distributed DBMSs were developed. Like H-Store/VoltDB, these DBMSs were designed to provide the same scalable performance of NoSQL systems for OLTP workloads while still maintaining the ACID guarantees of a traditional database system. In response to the NoSQL moniker, modern OLTP DBMSs are colloquially referred to as *NewSQL* systems [23].[2,3]

Along with H-Store, Clustrix is one the oldest NewSQL DBMSs [1]. Its architecture is similar to earlier distributed, disk-oriented DBMSs, such as Gamma [70] and NonStop SQL [212], except that it uses distributed multi-version concurrency control. Up until 2013, Clustrix was only available on proprietary hardware.

NuoDB [7] is another shared nothing, disk-oriented DBMS that also uses multi-version concurrency control. NuoDB's architecture is much different than other systems. All components of a database (e.g.,

[1] The NoSQL community argues that the sobriquet should be interpreted as "Not Only SQL", since some of these systems do support some variant of SQL.

[2] Some companies have used the NewSQL label for any OLTP database technology created after 2008, such as new storage engines or indexes for single-node DBMSs. We will only discuss the NewSQL systems that are new distributed DBMS architectures.

[3] The authors of this dissertation believe that H-Store was one of the first NewSQL systems.

tables, indexes, meta-data, etc.) are split into "atoms" that are spread out across a cluster [170]. These atoms then migrate to nodes based on query access patterns. Like SDD-1, each node in NuoDB is either a storage manager or a transaction executor (or both). When an executor node processes a query, it must retrieve a copy of the atoms that have that data that the query needs to access from a storage node. Before a transaction can commit, the atoms notify their other copies of any modifications and then resolve potential conflicts. Overtime, the atoms that are used together in the same transaction will end up on the same executor node. Thus, NuoDB can potentially reach the same optimal database design that Horticulture generates for H-Store.

VMWare's SQLFire [9] splits databases into main memory partitions stored across a cluster. It is similar to H-Store in that uses a decentralized transaction coordinator and supports Java-based stored procedures. The key difference, however, is SQLFire by default does not provide any isolation guarantees for transactions. For stronger isolation levels, it uses row-level read/write locks with 2PC.

MemSQL [4] is a distributed, main memory DBMS that targets "real-time" analytical workloads. It is similar to Hekaton in the execution engine at each node uses lock-free data structures with multi-version concurrency control. It does not support strongly consistent, serializable multi-node transactions like H-Store.

Finally, after starting the NoSQL movement with their BigTable system [54], Google has developed their own NewSQL system, called Spanner [63]. Spanner is as significant advancement in distributed transaction processing because it is the first large-scale system to support strong transaction consistency over geographically distributed data clusters. It uses two-phase locking with timestamp-based ordering that is similar to H-Store, where transactions wait for a period of time to see if any other transaction arrives with a lower id. They achieves this global ordering by using GPS devices and atomic clocks installed in each rack that allows each cluster to have a synchronized clock.

7.2 Database Design

The most notable advancements in automatic database design come from two commercial database vendors: Microsoft's SQL Server AutoAdmin [17, 18, 55, 57, 166] and IBM's DB2 Database Advisor [188, 231, 232]. We limit this discussion to the prior work that is relevant for distributed DBMSs.

7.2.1 Database Partitioning

A database table can be horizontally divided into multiple, disjoint fragments whose boundaries are based on the values of one (or more) of the table's columns (i.e., the *partitioning attributes*) [100, 230]. The DBMS assigns each tuple to a particular fragment based on the values of these attributes using either range partitioning or hash partitioning. Related fragments from multiple tables are combined together into a partition [90, 176]. Alternatively, a table can be replicated across all partitions. This is different than replicating entire partitions for durability and availability. Replication is useful for read-only or read-mostly tables that are accessed together with other tables but do not share foreign key ancestors.

The major differences amongst previous database partitioning approaches are in (1) selecting the candidate partitioning attributes [231] and (2) the search process used to find the optimal partitioning scheme. The former examines a sample workload and represents the usefulness of the candidate attributes extracted from the queries in auxiliary structures [17, 50, 188, 230]. Further heuristics can then be applied to prune this

candidate set [231] or combine attributes into multi-attribute sets [18]. The partitioning attributes for each table are selected using either an exhaustive search [230, 231], greedy search [188], or an approximation [66] algorithm that identify which of these candidates provide the best performance for the DBMS. This process usually occurs off-line: the partitioning algorithm compares potential solutions using a *cost model* that estimates how well the DBMS will perform using a particular design for the sample workload trace without needing to actually deploy the database (Section 7.2.4).

The study done in [230] compares the performance versus quality trade-offs of different search strategies. This work was completed as part of IBM's DB2 Database Advisor initiative [188, 221, 231]. The initial work in [231] describes an exhaustive search algorithm that chooses attributes based on relative weights derived from how the tables are referenced in the queries in a sample workload. Their later work in [188] uses the DBMS's own query optimizer to generate relevant partitioning attributes and then employs a brute-force algorithm that ranks them based on the queries' estimated disk I/O.

Microsoft's AutoAdmin finds sets of candidate attributes for individual queries and then attempts to merge them based on the entire workload [18]. The design tool in [166] also employs a branch-and-bound algorithm for selecting whether to partition or replicated tables in Microsoft's SQL Server 2008 Parallel Data Warehouse. The lack of support for stored procedures, replicated secondary indexes, and temporal skew handling limit the effectiveness of [18, 166] for the enterprise OLTP applications that we consider.

Schism is an automatic partitioning tool for the RelationalCloud system that also seeks to minimize multi-partitioned transactions [66]. For a given database, Schism scans all of the tables and populates a graph containing a separate vertex for every unique tuple. An edge is created between two vertices if the tuples that they represent are co-accessed together in a single transaction in the sample workload. Schism then applies a min-cut algorithm on the graph to produce partition boundaries that minimize the number of edges that cross partitions (i.e., distributed transactions), while also attempting to evenly distribute the workload and data. This partition assignment is then passed to a machine learning tool that generates a decision tree that routes tuples to the proper partition at run time.

Schism does not generate a design that includes replicated secondary indexes or stored procedure routing. The range partitioning scheme produced by Schism is also problematic for shared-nothing, main memory database environments. For example, when Schisms fails to find a decision tree for the min-cut graph partitions, it falls back to using a single lookup table that explicitly maps every tuple to a partition. These tables are large for even moderate-sized databases [66], and because memory is the critical resource in a main memory database, it is unlikely that one would be able to store such a table at every single node. Thus, one or more nodes must be designated as the centralized coordinators for the cluster and all queries are required to pass through them. This approach is adequate for the environment assumed in [66], where all of the transactions' control code is executed at the application; these central coordinators simply act as routers to direct incoming queries from the application to the right partition. But these coordinators and the maintenance of their lookup table are an unnecessary performance bottleneck for short-lived transactions that are indicative of the high-performance OLTP applications targeted by H-Store [211].

Even when Schism is able to generate a decision tree, it is still susceptible to scalability problems when dealing with new tuples that are outside the decision tree's computed boundaries. These "unmappable" tuples

are temporarily assigned to a random spare node until the partitioning algorithm is executed again. Schism does not guarantee that mapping these tuples to temporary partitions is done so in a manner that minimizes multi-partition transactions and that running the partitioning algorithm again requires scanning the entire database in order to generate a new graph. Conversely, H-Store's hash-based partitioning approach avoids Schism's problems by assigning tuples to partitions according to the schema tree generated our partitioning algorithm: every tuple is assigned to the partition that is likely to have the data that it is co-accessed with based on workload history. Our LNS algorithm is also based on workload traces and does not require the entire database to be loaded first.

Other work has focused on partitioning in main memory DBMSs. The method in [25] generates non-uniform partition sizes to accommodate the start-up delay in multi-node full-table sequential scan queries for data residing in memory. The authors in [99] compare different partitioning techniques for a main memory system to reduce the amount of disk I/O needed to reload the database after a crash.

7.2.2 Secondary Index Selection

Several approaches exist for selecting the indexes that reduce the amount of I/O needed to execute queries in traditional, disk-based DBMSs [56, 102, 108, 124]. This is different than the problem of selecting a replicated secondary index, since they are designed to reduce the number of distributed transactions in a distributed DBMS. Selecting vertical partitions in a database, however, is similar to selecting secondary indexes. For example, the AutoPart tool identifies conflicting access patterns on tables and creates read-only vertical partitions from disjoint subsets that are similar to our secondary indexes [179].

7.2.3 Transaction Routing

For stored procedure routing on shared-nothing DBMSs, the authors in [169, 184] provide a thorough discussion of the static, decentralized scheme supported by H-Store. The affinity-based routing approach in [184] directs requests to the nodes with data that the transactions will need using a broad categorization. The approaches in [168, 181] automatically generate a more fine-grained classification based on the previously executed transactions.

7.2.4 Cost Models

Database design algorithms use cost models to estimate how many resources the DBMS will use to execute a particular query or transaction for a given database design [197]. Each model is categorized as either (1) an analytical model that approximates resource consumption based on heuristics [50, 231] or (2) a "real-world" model that leverage's the DBMS's internal query optimizer to calculate the estimated cost [55, 230]. Much of the literature on cost estimation for main memory DBMSs is for single-node systems [58] or do not consider workload skew [156, 157].

7.3 Predictive Modeling

Modeling workloads using machine learning techniques is a well-known approach for extracting information about a database system [229]. To our knowledge, however, our work is the first to generate intra-transaction execution models (i.e., modeling what queries a transaction executes rather than simply what transactions

were executed) and use them to optimize the execution of individual transactions in a distributed database environment. Previous approaches either model workloads based on individual queries or sets of transactions in order to (1) manage resource allocation or (2) estimate future actions of other transactions.

In the former category, the Markov models described in [119, 120] are used to dynamically determine when the workload properties of an application have changed and the database's physical design needs to be updated. The techniques proposed in [80] identify whether a sample database workload is either for an OLTP- or OLAP-style application, and then tunes the system's configuration accordingly. The authors in [101] use decision trees to schedule and allocate resources for long running OLAP queries.

The authors in [194] generate Markov models that estimate the next query that an application will execute based on what query it is currently executing and then pre-fetches that query if there are enough resources available. Similarly, the authors in [78] use Markov models to estimate the next transaction a user will execute based on what transaction the DBMS is executing now. The work described in [228] does use Markov models based on queries much like ours, but their models are designed to identify user sessions across transactional boundaries and to extract additional usage patterns for off-line analysis purposes.

7.4 Speculative Execution

There are several techniques that are similar to the speculative execution methods that we propose in Chapter 6. These other approaches also seek overcome the overhead of distributed transactions. The differences are in the assumptions about workloads and in how to derive the information needed to make runtime decisions.

7.4.1 Speculative Transactions

The OCC technique from the seminal work in [15, 138] executes any transaction whenever a distributed transaction is stalled. The DBMS tracks transactions' read/write sets and checks for conflicts at commit time. The original work in [138] assumes that such conflicts are rare, and thus few transactions will be aborted. But for modern workloads targeted by NewSQL systems, this assumption is often incorrect. The work [189] in shows that OCC still performs better than two-phase locking in applications with a high frequency of transaction aborts. H-Store takes a pessimistic approach [175] and avoids conflicts in order to minimize the verification overhead at commit time.

Escrow transactions [173] allow transactions to update a record without needing to first acquire the lock for that record. This is similar to how we allow speculative transactions to execute at a partition without first acquiring its lock. The key difference, however, is that escrow transactions only work on single rows with numeric values and are not generalized for arbitrary data types.

Calvin [218] lacks mechanisms to determine this information automatically and thus requires the application to annotate each request. Furthermore, Calvin is unable to change execution order at runtime to improve cache locality. One way to support out-of-order execution of transactions is to use a more relaxed consistency model [187], but this is unacceptable in many OLTP applications. The coordination protocol in [37] allows out-of-order transaction execution while maintaining serializability.

Compensable transactions enable the DBMS to undo operations and correct the database in order to compensate for an error or conflict. They can be used to speculatively execute other transactions during long

running transactions [44].

7.4.2 Speculative Queries

Prefetching is a well-studied technique used in many aspects of computing systems, including CPU architectures and file systems. Speculator [167] enables the OS to speculatively execute processes while ensuring correct results by requiring that a speculative process wait until all previous processes that could affect it have finished. In the event that the process that speculative processes are waiting for fails, then the speculative process are restarted from checkpoints. QuickMine [202] predicts the blocks that a running transaction is likely to need and then prefetches them from the storage device.

Previous work has explored prefetching queries in DBMSs across transactional boundaries to compensate for high disk latency [200]. The authors in [194] generate Markov models that prefetches the next query that an OLAP application will execute based on what query the DBMS it is currently executing. SCOUT [214] prefetches data for a visualization tool based on the intrinsic structure of the object being viewed. Infobright [41] uses prefetching to retrieve and decompress data needed for future queries. Our technique is the first work that we are aware of that applies query prefetching to improve the response times of distributed transactions.

Chapter 8

Future Work

We now discuss several extensions to our work presented in this dissertation.

8.1 Distributed Transaction Optimizations

The techniques presented in Chapters 4 and 5 seek to minimize the total number of in a given application. In many workloads, however, distributed transactions are unavoidable in a large-scale, distributed DBMS [113].

These DBMSs use a commit protocol to ensure that operations happen atomically. The most used of these systems use two-phase commit (2PC) [35, 93]. The conventional wisdom, however, is that 2PC is a slow operation [142], and many optimizations [193] and variations [111, 136] have been proposed.

Rather than replace the 2PC protocol, we seek to exploit certain inherent properties of distributed transactions in OLTP workloads to increase the throughput of the overall system while also reducing latency of distributed transactions. Thus, we propose two optimizations for multi-partition transactions in a distributed DBMS: (1) *transaction batching* and (2) *transaction splitting*. These techniques are optimistic (i.e., they assume that most transactions will not abort and that hardware problems are infrequent), but are still fault tolerant and resilient to unexpected node or network failures.

8.1.1 Transaction Batching

With transaction *batching*, the DBMS will reduce both the amount of time spent initializing a distributed transaction across multiple partitions and the amount of time that a partition remains idle. The main idea is to amortize these costs across multiple requests and allow for multiple distributed transactions that need to access the same partitions to execute in parallel. We can do this by batching together transactions that either have the same or different base partitions.

In same-partition batching, the transaction coordinator at a particular node will execute multiple transactions at the same base partition in succession using the same partition leases. This means that there is only a single initialization message and a single "finish" 2PC (or abort) message for multiple transactions. When one transaction completes, the next one in the batch is immediately executed at the same partition. This is similar to the speculative lock-inheritance technique proposed for Shore-MT [126].

With cross-partition batching, the transaction coordinator will combine partition lock leases for commutative transactions that are executing on disparate partitions. This allows for a distributed transaction to execute on each of these partitions concurrently. The queries for each of the combined transactions will be interleaved with the local transaction's queries.

8.1.2 Transaction Splitting

The main idea of *transaction splitting* is to divide a distributed transaction into multiple single-partition transactions that each perform an independent piece of the transaction that does not need to be coordinated across the cluster. As opposed to the batching optimization described above, which seeks to minimize the total number of coordination messages for the entire workload, this splitting technique seeks to minimize the number of coordination messages for a single transaction.

Consider a transaction t_1 for a stored procedure that always executes two queries. The procedure is given as its input a unique, non-partitioning identifier that has a one-to-one mapping to the primary key for a table. The first query is a full-table scan that searches for the primary key of a table using the non-partitioning identifier. This query is broadcast to all nodes, executed in parallel, and the results from each partition are sent back to the transaction's base partition. Then the procedure submits a second query that updates one record using the primary key that was returned in the first query. All other partitions will remain idle until they receive the "prepare" 2PC message. Thus, the DBMS must execute the stored procedure's control code (cf. Section 3.3) for each transaction invocation on a random node that the client sent the request to because it is not possible for the coordinator to know what partition the transaction needs until after it executes the first query.

Instead of running the control code at a single location and then broadcasting queries out to remote partitions, we can instead split a distributed transaction into multiple single-partition transactions that are each executed on one partition. As each transaction executes at each partition, the execution engine forces the query planner to direct all query requests to the local partition. Using the above example, the first query will attempt to retrieve the primary key value based on the transaction's input parameter (i.e., the unique, non-partitioning identifier). Only one partition will have a record that matches that value, and thus all other partitions will return an empty result.

8.2 Many-Core Concurrency Control

When all of the transactions are single-partitioned irrespective to the number of partitions, then H-Store's architecture model can scale infinitely. Note, however, that scaling out the database over more partitions will improve the overall throughput of the system, but will not increase the execution speed of each transaction (i.e., the time that it takes for the DBMS to process the transaction from beginning to end, excluding setup time and queue delays). This is because the speed of a single-partition transaction is limited to the clock speed of a single CPU core. Because H-Store avoids any shared data structures, a single core must execute all of the components needed to process transaction at a single partition, including the control code executor, the query planner, and the storage manager.

In prior decades, one could rely on the expectation that the clock speeds of a single-core increases year

after year. This trend, however, has tapered off in recent years; the current hardware trend from chip manufacturers is to increase the number of cores on a single CPU rather than increase the clock speed. Furthermore, it is also speculated that although future CPU architectures will have many more cores than in current chip designs, many of these cores will be specialized for a specific type of computation, rather than being general-purpose core [36]. This means that only a subset of the CPU's cores could be used to execute transactions under the current H-Store model. Since the only way to increase performance in H-Store is to increase the number of partitions, an administrator will have to provision more machines for their database cluster.

There are several drawbacks, however, of increasing performance by only adding more partitions. Many OLTP applications have a small number of distributed transactions can never be single-partitioned unless the database only has one partition. Thus, scaling out horizontally by just adding more cores (and as a result more partitions) will have diminishing returns for many workloads. As the number of partitions increases, the likelihood that the data that a transaction needs to access is stored in another partition increases: what was once a single-partition transaction in a smaller cluster now becomes a distributed transaction in the larger cluster, which means that such a transaction is executed with heavyweight concurrency control.

Other scaling problems can arise if a main-memory DBMS is deployed on a machine with several hundreds or even thousands of cores. As the number of partitions increases on a single-machine and more transactions are executed concurrently, the memory controller for that node will become the main bottleneck [39]. Non-uniform memory access architectures attempt to alleviate this problem by using separate physical memory modules per CPU socket, but are still susceptible to cache contention problems for intensive workloads that utilize all workloads evenly. Administrators will have to decide to leave some cores idle and provision more nodes to the cluster, which increases both the management and energy costs

To overcome this limitation, we plan to explore the limitations of existing concurrency control schemes using a distributed parallel simulator for multi-core architectures. We will measure the lock contention and overhead on a simulated platform of thousands of cores. We will then develop new lock-free execution model that allows transactions to execute concurrently on the same logical partition [116, 143].

8.3 Database Design

We are extending our Horticulture tool from Chapter 4 to generate database designs for different types of systems. For example, we are working adapting our tool for document-oriented NoSQL DBMSs to select the optimal sharding and index keys, as well to denormalize schemas. This work shows that Horticulture's LNS-based approach is adaptable to many different systems just by changing the cost model. We modified our cost model for NoSQL systems to estimate the number of disk operations per operation [230] and the overall skew using the same technique presented in Section 4.4.2. Because these systems do not support joins or distributed transactions, we do not need to use our coordination cost estimation. Supporting database partitioning for a mixed OLTP and analytical workloads in Horticulture is another interesting research area. A new cost model would have to accommodate multiple objectives, such as improving intra-query parallelization in analytical workloads while also satisfying service-level agreements for the front-end workload. As our work continues on automatic database design for distributed systems, we plan to integrate a commercial constraint programming solver into Horticulture.

8.4 Database Reorganization & Elasticity

Previous studies have shown the important of placing data that is used together often in transactions physically closer together [183]. This reduces the amount of cross-partition communication. Another problem in H-Store is that it is highly susceptible to skew [181]. If there is one hot record at a partition, then transactions will get backed up in the queue for that partition's execution engine. Thus, that partition will become overloaded while the other partitions are under-utilized.

Given this, we are interested in how to identify these affinities and hotspots in the system at runtime. We plan to apply our data partitioning algorithms for on-line database reorganization. When the workload properties of an existing DBMS installation changes, our algorithms will generate new designs that adjust to changes in workload and data skew [223]. This will allow H-Store to identify overloaded partitions and then automatically migrate data to either other existing partitions or bring new partitions on-line. We are also developing data placement algorithms that assign the location and sizes of partitions to particular nodes in a cluster [176]. Generating an optimal placement strategy can improve the performance of a distributed transaction by increasing the likelihood that any "non-local" partition is located on the same node.

8.5 Predictive Transaction Modeling

Representing transactions with Markov models in the manner discussed in Chapters 5 and 6 is also applicable to several other research problems in distributed OLTP systems. We plan on extending our models to include additional information about transactions, such as their resource usage and execution times. This information could then be used for admission control or the intelligent scheduling of transactions based on the results of the initial path estimates [101]. For example, the execution states in a model could also include the expected remaining run time for a transaction. By examining the relationships between queries and the procedure parameters, we can discover commutative sets of queries that could then be pre-fetched if the transaction enters some "trigger" state [194]. Similarly, the models could also identify sets of redundant queries in a transaction that could automatically be rewritten and grouped into a smaller batch.

We are currently investigating techniques for the automatic reorganization of on-line H-Store deployments in order to respond to changes in demand and workload skew. We plan on leveraging our models' ability to quickly compare the expected execution paths of transactions with the actual execution properties of the current workload. Such automatic changes include scaling up the number of partitions in the system or repartitioning the database.

The obvious addition to our speculative execution scheme presented in Chapter 6 is to add support for distributed transactions. This is will be part of our effort for dynamically constructing batches of transactions at runtime, as described in Section 8.1.1.

We are interested in improving support for workloads whose properties are not easily captured by our Markov models. This includes allowing the DBMS to prefetch remote partition queries whose input parameters are derived from the output of previous queries in the transaction. The DBMS can add triggers automatically to a procedure's control code so that queries are dispatched once the results of those earlier queries are received. Similar to "runahead" execution [165], the DBMS could also make all query requests non-blocking and then resolve their results during 2PC so that transactions never stall at **SP1**. Alternatively,

if only some of the input parameters are known for the WHERE clause of a remote partition query, then the DBMS can prefetch a "less precise" query and return the results to the transaction's base partition. Once the full query is known, then the DBMS can resolve locally what tuples the query actually needs.

8.6 Larger-than-Memory Databases

The fundamental problem with main memory DBMSs is that their improved performance is only achievable when the database is smaller than the amount of physical memory available in the system. If the database does not fit in memory, then the operating system will start to page virtual memory, and main memory accesses will cause page faults. Because page faults are transparent to the user, in this case the main memory DBMS, the execution of transactions is stalled while the page is fetched from disk. This is a significant problem in a DBMS, like H-Store, that executes transactions serially without the use of heavyweight locking and latching [203]. Because of this, all main memory DBMSs warn users not to exceed the amount of real memory [219]. If memory is exceeded (or if it might be at some point in the future), then a user must either (1) provision new hardware and migrate their database to a larger cluster, or (2) fall back to a traditional disk-based system, with its inherent performance problems.

To overcome these problems, we are developing a new architecture for main memory DBMSs that we call *anti-caching*. In a DBMS with anti-caching, when memory is exhausted, the DBMS gathers the "coldest" tuples and writes them to disk with minimal translation from their main memory format, thereby freeing up space for more recently accessed tuples. As such, the "hotter" data resides in main memory, while the colder data resides on disk in the anti-cache portion of the system. Unlike a traditional DBMS architecture, tuples do not reside in both places; each tuple is either in memory or in a disk block, but never in both places at the same time. In this new architecture, main memory, rather than disk, becomes the primary storage location. Rather than starting with data on disk and reading hot data into the cache, data starts in memory and cold data is evicted to the anti-cache on disk.

8.7 Workload Expansion

We are interested in expanding the scope of the workloads supported in H-Store. Normally organizations stream data out of a front-end system using an ETL process and store it in a data warehouse. They can then execute longer running, analytical queries on this back-end system without affecting the front-end OLTP DBMS. But because there is a delay in getting transferring this information from one DBMS to the other, the OLAP system could be viewing potentially stale data. Thus, we are interested in supporting what is known as "real-time" analytics in a high-performance DBMS like H-Store. The goal is to allow users to perform OLAP queries directly in the front-end DBMS, thereby reducing the time to find answers. Related to this, we are working on adding support for continuous queries (i.e., stream processing) as first entities in the system. The DBMS will execute these more complex operations alongside the OLTP transactional workload without affecting the performance of the front-end application. We plan to investigate different strategies for supporting these workloads, including using snapshots like in HyPer [134], or maintaining multiple data stores inside of the same DBMS like in SAP HANA [84, 199] and OctopusDB [77].

To support more complex queries in H-Store's query planner, we plan to investigate an alternative approach that is used by MongoDB [5]. In that system, instead of using a cost-based query optimizer [197], the DBMS generates *all* possible plans for a query and tries them each out to determine which one is the best. At runtime, the system will select a random plan for a query and keep track of how long it takes to complete. Once enough samples are collected, the DBMS will then choose the query plan with the lowest run time. The information gained about query plans can be written into the DBMS's internal catalog so that if the DBMS is restarted it can use the best query without having to run trials first.

Lastly, we will develop a new DBMS beyond H-Store that supports high-performance transaction processing for non-partitionable workloads. The TPC-E benchmark (cf. Section A.6) is an example of such a workload [215].

8.8 Non-Volatile Memory

Just as changes in the number of cores per CPU and the reduction of memory prices enabled H-Store's departure from a traditional DBMS architecture, the onset of non-volatile memory (NVM) devices will require a re-evaluation of the dichotomy between memory and durable storage. The next phase in database system development will focus on using NVM-based storage [190] – also referred to as storage class memory [43]. The promise of this new nanoscale technology is that it will have the read/write performance of DRAM, but with the persistency and durability of SSDs. There are several emerging technologies that will compete in the NVM space [43, 190]. These devices promise to overcome the disparity between processor performance and DRAM storage capacity limits that encumber data-centric applications.

We believe that there are several avenues for research with these NVM devices. Foremost is the potential of resistive NVM devices to dynamically change their storage area into executable logic gates. We plan to explore integrating machine learning components into a DBMS to automatically enable these executable configurations in NVMs. For example, if a user is sequentially scanning large segments of DNA data, then the DBMS could migrate the processing logic for identifying interesting sequences from the application down into the NVM. Once again, the research challenge lies in how to enable the DBMS to identify when it is appropriate to perform this optimization and to understand the trade-offs. We will also explore partitioning techniques that consider locality when storing data in these NVM devices, since the difference between accessing data on the same CPU socket versus a different socket in the same node will be significantly greater than it is now [183]. We believe that a NVM-based system that account for these types of problems will greatly outperform existing DBMSs because they do not need to copy data from storage to the CPU.

Chapter 9

Conclusion

In this dissertation, we presented H-Store, a new DBMS that is designed for main memory storage with minimal concurrency control overhead in OLTP applications. H-Store executes transactions efficiently using stored procedures without fine-grained locks or latches. It splits a database into in-memory partitions that are managed by single-threaded execution engines. Each partition can contain a disjoint subset of a table or a copy of a replicated table. When one these engines executes a transaction, that transaction never stalls waiting to acquire a lock held by another transaction or waiting for additional input from the application. As shown in Section 3.9, H-Store performs up to 25× faster on a single node than traditional, disk-based DBMSs. Although H-Store is not a general purpose DBMS, and thus will not perform well for all workloads, there are a sizable number of applications with the workload characteristics targeted by our system (cf. Appendix A).

Many OLTP databases, however, are larger than the amount of memory available on a single node. For these applications, the database is partitioned across a cluster of shared-nothing nodes. The main drawback of a distributed deployment is that transactions that used to only access data at a single node may now need to access multiple nodes. The network communication overhead for these multi-node transactions can cause a distributed, main memory DBMS to perform no better than a single-node system. But it is non-trivial to scale a distributed, main memory DBMS to execute larger volumes of transactions effectively; the recent trend in scaling DBMSs without ACID is insufficient for OLTP applications that need strong consistency guarantees.

To overcome this problem, we presented three different optimization techniques in this dissertation for high-performance transaction processing in a distributed, main memory DBMS. First, we showed in Chapter 4 how to generate a physical design that deploys a database across nodes in such a way that minimizes the number of partitions that each transaction needs to access. If all of the data that a transaction needs is stored within the same partition, then the DBMS executes that transaction using a lightweight concurrency control scheme that does not require it to coordinate with other nodes in the cluster. These designs also minimize the amount of skew in the workload so that no one partition becomes overloaded.

In order to utilize this lightweight concurrency control scheme, the DBMS needs to know certain information about each transaction's expected runtime behavior. This information includes the number of partitions that they need to access or whether a particular transaction could abort. The system derives this information

from developer-written annotations in the application, but they are too coarse-grained and thus may be inaccurate. Another approach is to blindly execute the transaction as single-partitioned and then restart it when it deviates from this expectation. The problem with this approach is that the system's throughput decreases on larger cluster configurations due to transactions having to continually restart. Hence, in Chapter 5 we showed how to incorporate a machine learning framework in the DBMS to predict the execution behavior of transactions before they start running. When a transaction request comes into the system, the DBMS uses this framework to identify whether it should be executed as a fast, single-partition transaction or a slow, multi-partition transaction.

Our first two techniques allow the DBMS to avoid distributed transactions. They do not improve the throughput or latency of the system when distributed transactions are actually running. Thus, in Chapter 6, we identified the points during a distributed transaction's lifetime that the DBMS will stall because of cross-partition communication. We then described how to leverage the predictions generated by our machine learning framework from Chapter 5 to schedule speculative tasks at partitions during these stall points. We showed how the system can interleave single-partition transactions at a partition without violating the serializability guarantees of the system. We also showed how to identify queries that the DBMS can prefetch before a distributed transaction starts so that it does not need to wait for their results at runtime.

All together, the work described in this dissertation allow a distributed, main memory DBMS to support transactional workloads beyond what a single node system can support.

Appendix A

OLTP Benchmarks

We now provide a more thorough description of the OLTP benchmarks used in our evaluations in this dissertation. These benchmarks were ported to H-Store in a good faith to follow the original spirit of each benchmark's specification, but may differ in the implementation details.

A.1 AuctionMark

AuctionMark is a OLTP benchmark by Brown University and a well-known online auction web site [104]. The benchmark is specifically designed to model the workload characteristics of an online auction site running on a shared-nothing parallel database. It consists of 16 tables and 14 stored procedures, including one procedure that is executed at a regular interval to process recently ended auctions. On particularly challenging aspect of AuctionMark for partitioning algorithms is that there are several tables of varying size that are "read-mostly". If these tables are replicated at all partitions, then many of the transactions can execute as single-partitioned. But depending on the rate in which these read-mostly tables are updated, the benefits gained from allowing some transactions to execute as single-partitioned are offset by the multi-partition transactions that update the replicated tables. Partitioning the tables by the `ITEM` id and the seller's `USER` id maximizes the number of single-partitioned transactions while also minimizing the amount of skew. The user-to-item ratio follows a Zipfian distribution, meaning that there are a small number of users that are selling a large portion of the total items. Thus, if the tables are only partitioned on the seller's id, then both the `USER`-centric data and transactions would overload partitions.

A.2 SEATS

The SEATS benchmark models an airline ticketing system where customers search for flights and make reservations [105]. It consists of eight tables and six stored procedures. Finding a good design for SEATS is more challenging than TATP and TPC-C because of non-read-only table replication candidates and data access patterns. For example, the non-uniform distribution of flights between airports creates imbalance for large cluster sizes if the database was horizontally partitioned by airport-related columns.

	Tables	Columns	Foreign Keys	Indexes	Procedures	Read-Only Txns
AuctionMark	16	123	41	14	9	55%
SEATS	8	197	16	18	6	45%
SmallBank	3	6	2	4	6	15%
TATP	4	51	4	7	7	40%
TPC-C	9	92	24	3	5	8%
TPC-E	33	191	50	43	12	77%
Voter	3	9	1	3	1	0%

Table A.1: Profile information for the benchmark workloads.

A.3 SmallBank

This workload models a banking application where transactions perform simple read and update operations on customers' accounts [45]. All transactions involve a small number of tuples that are retrieved using primary key indexes. The transactions' access patterns are skewed such that a small number of accounts receive most of the requests. We extended the original SmallBank implementation to include a transaction that transfers money from one customer's account to another [83].

A.4 TATP

The TATP benchmark (formerly Telecom One or TM1) is a newer OLTP testing application that simulates a typical caller location system used by telecommunication providers [226]. The benchmarks consists of four tables, three of which are foreign key descendants of the root `SUBSCRIBER` table. All procedures reference tuples using either `SUBSCRIBER`'s primary key or a separate unique identification string. Those stored procedures that are given this primary key in their input parameters are always single-partitioned, since they can be immediately directed to the proper partition. Other procedures that only provide the non-primary key identifier have to broadcast queries to all partitions in order to discover the partition that contains the `SUBSCRIBER` record that corresponds to this separate identification string.

A.5 TPC-C

The TPC-C benchmark is the current industry standard for evaluating the performance of OLTP systems [216]. It consists of nine tables and five stored procedures that simulate a warehouse-centric order processing application. All of the stored procedures in TPC-C provide a warehouse id as an input parameter for the transaction, which is the ancestral foreign key for all tables except `ITEM`. Approximately 90% of the `ORDER` records can be processed using single-partition transactions because all of the corresponding `ORDER_LINE` records are derived from the same warehouse as the order. The other 10% transactions have queries that must be re-routed to the partition with the remote warehouse fragment. Thus, the two main challenges for automatically partitioning the TPC-C database is (1) deciding what warehouse id column to use for partitioning `ORDER_LINE` and (2) identifying that the `ITEM` table should be replicated. For the former, the `ORDER_LINE` table is co-accessed with the `STOCK` table more often that it is with the `ORDERS` table, but the supply warehouse id of the `STOCK` table is not considered the "local" warehouse of the order.

A.6 TPC-E

Since TPC-C is 20 years old, the TPC-E benchmark was developed to represent more modern OLTP applications [215]. The TPC-E schema contains 33 tables with a diverse number of foreign key dependencies between them. It also features 12 stored procedures, of which ten are executed in the regular transactional mix and two that are considered "clean-up" procedures that are invoked at fixed intervals. These clean-up procedures are of particular interest because they perform full-table scans and updates for a wide variety of tables. It is very challenging to find the optimal design for the TPC-E benchmark for several reasons. Foremost is that unlike the TPC-C and AuctionMark benchmarks, where the schema trees are long and narrow, the TPC-E schema tree is short and wide. This means that many of the tables have foreign key dependencies to other disparate tables, which creates many conflicting partitioning candidates. Many of the stored procedures also have optional input parameters that cause transactions to execute different sets of queries based on what parameters are given at run time. For example, the `DataMaintenance` procedure will execute one out of 12 possible sets of queries based on the name of the table passed in to the transaction. This means that is difficult to find a static partitioning parameter that directs these procedures to the proper partition for all cases.

A.7 Voter

The Voter benchmark 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. There are a fixed number of contestants in the database. The majority of the Voter's workload are transactions that update the total number of votes for a particular contestant. The DBMS records the number of votes made by each user based on their phone number; each user is only allowed to vote a fixed number of times. A separate transaction is periodically invoked in order to display the vote totals while the show is broadcast live.

Appendix B

Query Plan Operators

We now present an overview of the query plan operators that H-Store supports. H-Store supports most of the SQL-92 specification. In some cases, we chose to implement non-standard functionality used in Postgres and MySQL (e.g., `LIMIT`) rather than follow the SQL standard. But the operators listed here are independent to the actual SQL dialect that the DBMS supports.

We have grouped the 16 operators together here based on their function.[1] Each non-leaf operator takes in one or more input tables that are generated from another operator, and produces one and only one output table. An operator can also have additional configuration options that are added by the DBMS's query planner (e.g., the number of tuples to include the output for the LIMIT operator). We refer the interested reader to Section 3.5 for a description of how H-Store generates query plans based on these operators.

B.1 Scan Operators

These operators are used to retrieve tuples from database's tables. With the exception for INSERT queries, these two operators are always the leaf vertexes in the query plan tree.

SEQSCAN: This will cause the engine to perform a complete sequential scan on a single table. The operator examines each individual tuple in the table and emits those tuples into its output table that satisfy the predicates in the query's WHERE clause. The scan will stop until all tuples are examined or if there is an embedded LIMIT operator for this operator that has been satisfied (which ever comes first). Since tuples are stored in unsorted order, there is no terminating predicate that will cause the operator to stop scanning.

INDEXSCAN: Scan the tuples for a table through one of its indexes. The operator is provided with a search key used to jump to a location in the index and then it will iterate over the index's entries in sorted order. Tuples are emitted to the output table if they satisfy the predicates in the query's WHERE clause. The search halts when the query's stop predicate evaluates to false or if there is an embedded LIMIT operator for the this operator that has been satisfied (which ever comes first). H-Store does not

[1] As of 2013, VoltDB still uses all 16 of these original operators from H-Store plus two additional ones for performing scans on materialized views and computing faster count aggregates on indexes.

currently support covering indexes, so the operator will always follow the pointer from the indexes to retrieve the full tuple contents.

B.2 Join Operators

H-Store currently supports two variants of the canonical nested-loop join algorithm [185]. Although other more optimized and cache-consious implementations exist, they would provide little improvements for the workloads that we used in this dissertation because the scope of joins are small.

NESTLOOP: Given two input tables, the operator designates the first one as the "outer" table and the second one as the "inner" table. For each tuple in the outer table, the engine scans every tuple in the inner table and evaluates the operator's join predicate. If this predicate evaluates to true, then the tuple is added to the operator's output table.

NESTLOOPINDEX: This operator takes in one input table that it designates as the "outer" table. The "inner" table comes from an embedded INDEXSCAN operator. For each tuple in the outer table, the engine extracts the values for the columns that correspond to the join predicate and then uses the INDEXSCAN to find the tuples in the inner table that match that key. This is the most common join operator used for all of the workloads in Appendix A.

B.3 Modifying Operators

These operators are used to modify the contents of a database.

INSERT: This operator inserts all of the tuples in its input table into a target table in the database. As each tuple is inserted, the engine will update the table's indexes and write an entry in the transaction's undo log (cf. Section 3.6.3).

DELETE: Given an input table, this operator will delete all the entries in that input table from a target table in the database. The DELETE operator checks whether its input table is produced from a PROJECTION operator executing on the same partition. If it is, then the tuples in that input table will contain a single column that corresponds to their 4-byte offset. The operator uses that offset to retrieve the tuple's location in memory so that it can be deleted directly from the table. Note that the system still has to traverse all of the target table's indexes using the values of each tuples' fields to remove them from the indexes.

UPDATE: The input table for this operator contains the primary keys of the tuples to be updated and the columns with the new values for those tuples. This input is generated from one of the scan operators. The operator iterates over each record and writes the new values directly into the tuple's location in memory and updates any indexes for the target table.

B.4 Data Operators

These operators are for performing certain actions on transient data (i.e., output tables generated from other operators, or data from the transaction's control code).

MATERIALIZE: The MATERIALIZE operator is used to convert the query invocation's input parameters into tuples. These tuples are then copied into the operator's output table. This is typically used with the INSERT operator.

RECEIVE: This is used to combine one or more input tables from a corresponding SEND operator. This is the similar to the UNION except that the data comes from transaction coordinator (rather than from an output table from another operator in the same execution engine). The RECEIVE operator may execute on a different partition than from where the SEND operator executed, but its input data will always be sent from the transaction's base partition coordinator. This operator is only used in multi-partition query plans (cf. Section 3.5.2).

SEND: This instructs the execution engine to send the output table from this operator back to the transaction's base partition. This output table will be used as input for a corresponding RECEIVE operator. The SEND operator always emits its input tables directly as its output table. This operator is only used in multi-partition query plans (cf. Section 3.5.2).

UNION: This operator combines one or more input tables from other operators running in the same partition. The tuples from each input table are merged in unsorted order.

B.5 Output Operators

The following operators are used to control the contents of the output tables for a query plan. These are typically invoked at the top of the query plan's tree.

AGGREGATE: This operator will compute one or more aggregates on the target columns of its input table. H-Store supports the MIN, MAX, SUM, COUNT, and AVERAGE aggregate functions. This operator also supports the GROUP BY operation.

DISTINCT: This prunes out the non-distinct tuples from the operators input table. For a given set of columns, the operator examines its input table and constructs a composite key for each tuple. If that tuple does not already exist in the set of previous keys seen for that invocation, then the key is added to that set and the tuple is added to the operator's output table. This operator can be embedded in the SEQSCAN and INDEXSCAN operators.

LIMIT: Given an input table, this operator will only emit a certain number of tuples from that input table into its output table. The number of tuples to include in the output table can come from a constant value in the SQL statement or from an input parameter. H-Store's LIMIT operator also supports offsets so that the first n tuples are skipped. This operator can be embedded in the most commonly used operators for OLTP queries (i.e., SEQSCAN, INDEXSCAN, and ORDERBY).

ORDERBY: This operator sorts its input table based on one or more of its columns. It supports multiple sort columns in either ascending or descending direction.

PROJECTION: The PROJECTION operator has a special output column type that stores the tuple's 4-byte offset in the output table (instead of its full tuple). This is used for DELETE and UPDATE operators that execute on the same node.

Bibliography

[1] Clustrix. http://www.clustrix.com.

[2] Apache HBase. http://hbase.apache.org.

[3] H-Store. http://hstore.cs.brown.edu.

[4] MemSQL. http://www.memsql.com.

[5] MongoDB. http://mongodb.org.

[6] Mysql. http://www.mysql.com.

[7] NuoDB. http://www.nuodb.com.

[8] Redis. http://redis.io.

[9] VMware vFabric SQLFire. http://www.vmware.com/go/sqlfire.

[10] VoltDB. http://www.voltdb.com.

[11] Wikipedia MySQL Server Roles. https://wikitech.wikimedia.org/view/Server_roles.

[12] *The World in 2013: ICT Facts and Figures*. ITU, 2013.

[13] D. J. Abadi. *Query Execution in Column-Oriented Database Systems*. PhD thesis, MIT, 2008.

[14] D. J. Abadi, Y. Ahmad, M. Balazinska, M. Cherniack, J. hyon Hwang, W. Lindner, A. S. Maskey, E. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the borealis stream processing engine. In *CIDR*, pages 277–289, 2005.

[15] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. In *SIGMOD*, pages 23–34, 1995.

[16] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in sql databases. In *VLDB*, 2000. ISBN 1-55860-715-3.

[17] S. Agrawal, S. Chaudhuri, A. Das, and V. Narasayya. Automating layout of relational databases. In *ICDE*, pages 607–618, 2003.

[18] S. Agrawal, V. Narasayya, and B. Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *SIGMOD*, 2004. ISBN 1-58113-859-8. doi: http://doi.acm.org/10.1145/1007568.1007609.

[19] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *SOSP*, pages 159–174, 2007.

[20] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. Dbmss on a modern processor: Where does time go? VLDB, pages 266–277, 1999.

[21] P. M. G. Apers, C. A. van den Berg, J. Flokstra, P. W. P. J. Grefen, M. L. Kersten, and A. N. Wilschut. PRISMA/DB: A parallel, main memory relational DBMS. *IEEE Trans. on Knowl. and Data Eng.*, 4 (6):541–554, 1992.

[22] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, Apr. 2010.

[23] M. Aslett. How will the database incumbents respond to NoSQL and NewSQL? The 451 Group, April 2011.

[24] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System r: relational approach to database management. *ACM Trans. Database Syst.*, 1(2):97–137, June 1976.

[25] N. Bassiliades and I. P. Vlahavas. A non-uniform data fragmentation strategy for parallel main-menory database systems. In *VLDB*, pages 370–381, 1995. ISBN 1-55860-379-4.

[26] J. Baulier, P. Bohannon, S. Gogate, S. Joshi, C. Gupta, A. Khivesera, H. F. Korth, P. McIlroy, J. Miller, P. P. S. Narayan, M. Nemeth, R. Rastogi, A. Silberschatz, and S. Sudarshan. DataBlitz: A high performance main-memory storage manager. VLDB, pages 701–, 1998.

[27] P. Bernstein, J. Rothnie, J.B., N. Goodman, and C. Papadimitriou. The concurrency control mechanism of SDD-1: A system for distributed databases (the fully redundant case). *Software Engineering, IEEE Transactions on*, SE-4(3):154–168, 1978.

[28] P. Bernstein, D. Shipman, and W. Wong. Formal aspects of serializability in database concurrency control. *IEEE Transactions on Software Engineering*, 5(3):203–216, 1979.

[29] P. A. Bernstein and N. Goodman. Timestamp-based algorithms for concurrency control in distributed database systems. In *VLDB*, pages 285–300, 1980.

[30] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221, 1981.

[31] P. A. Bernstein and N. Goodman. Multiversion concurrency control - theory and algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, Dec. 1983.

[32] P. A. Bernstein and E. Newcomer. *Principles of Transaction Processing*. 2nd edition, 2009.

[33] P. A. Bernstein and D. W. Shipman. The correctness of concurrency control mechanisms in a system for distributed databases (SDD-1). *ACM Trans. Database Syst.*, 5(1):52–68, Mar. 1980. ISSN 0362-5915.

[34] P. A. Bernstein, N. Goodman, E. Wong, C. L. Reeve, and J. B. Rothnie, Jr. Query processing in a system for distributed databases (sdd-1). *ACM Trans. Database Syst.*, 6(4):602–625, Dec. 1981.

[35] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*, chapter 7. Addison Wesley Publishing Company, 1987. ISBN 0-201-10715-5.

[36] S. Blanas, Y. Li, and J. M. Patel. Design and evaluation of main memory hash join algorithms for multi-core cpus. SIGMOD, pages 37–48, 2011.

[37] S. Blott and H. F. Korth. An almost-serial protocol for transaction execution in main-memory database systems. In *VLDB*, pages 706–717, 2002.

[38] P. Bohannon, D. Lieuwen, R. Rastogi, A. Silberschatz, S. Seshadri, and S. Sudarshan. The architecture of the dalí main-memory storage manager. *Multimedia Tools Appl.*, 4(2):115–151, Mar. 1997.

[39] P. A. Boncz, S. Manegold, and M. L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 54–65, 1999. ISBN 1-55860-615-7.

[40] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez. Prototyping bubba, a highly parallel database system. *IEEE Trans. on Knowl. and Data Eng.*, 2(1):4–24, Mar. 1990.

[41] J. Borkowski. Increasing query speed with multithreaded data prefetching. *PARELEC*, pages 117–122, April 2011.

[42] A. J. Borr. Transaction monitoring in ENCOMPASS: reliable distributed transaction processing. volume 7 of *VLDB*, pages 155–165, 1981.

[43] G. W. Burr, B. N. Kurdi, J. C. Scott, C. H. Lam, K. Gopalakrishnan, and R. S. Shenoy. Overview of candidate device technologies for storage-class memory. *IBM J. Res. Dev.*, 52(4):449–464, July 2008. ISSN 0018-8646.

[44] M. Butler, T. Hoare, and C. Ferreira. A trace semantics for long-running transactions. In *Communicating Sequential Processes*, volume 3525 of *Lecture Notes in Computer Science*, pages 707–711. 2005.

[45] M. J. Cahill, U. Röhm, and A. D. Fekete. Serializable isolation for snapshot databases. SIGMOD, pages 729–738, 2008. ISBN 978-1-60558-102-6.

[46] M. J. Carey and M. Livny. Distributed concurrency control performance: A study of algorithms, distribution, and replication. VLDB, pages 13–25, 1988.

[47] M. J. Carey, D. J. DeWitt, M. J. Franklin, N. E. Hall, M. L. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O. G. Tsatalos, S. J. White, and M. J. Zwilling. Shoring up persistent applications. SIGMOD, pages 383–394, 1994.

[48] M. R. Casanova and P. A. Bernstein. A formal system for reasoning about programs accessing a relational database. *ACM Trans. Program. Lang. Syst.*, 2(3):386–414, July 1980. ISSN 0164-0925.

[49] R. Cattell. Scalable sql and nosql data stores. *SIGMOD Rec.*, 39:12–27, 2011.

[50] S. Ceri, S. Navathe, and G. Wiederhold. Distribution design of logical database schemas. *IEEE Trans. Softw. Eng.*, 9(4):487–504, 1983. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/TSE.1983.234957.

[51] S. K. Cha and C. Song. P*TIME: highly scalable OLTP DBMS for managing update-intensive stream workload. VLDB, pages 1033–1044, 2004.

[52] D. D. Chamberlin, M. M. Astrahan, M. W. Blasgen, J. N. Gray, W. F. King, B. G. Lindsay, R. Lorie, J. W. Mehl, T. G. Price, F. Putzolu, P. G. Selinger, M. Schkolnick, D. R. Slutz, I. L. Traiger, B. W. Wade, and R. A. Yost. A history and evaluation of system r. *Commun. ACM*, 24:632–646, October 1981. ISSN 0001-0782.

[53] A. Chan, S. Fox, W.-T. K. Lin, A. Nori, and D. R. Ries. The implementation of an integrated concurrency control and recovery scheme. SIGMOD, pages 184–191, 1982.

[54] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26:4:1–4:26, June 2008. ISSN 0734-2071.

[55] S. Chaudhuri and V. Narasayya. Autoadmin "what-if" index analysis utility. *SIGMOD Rec.*, 27(2): 367–378, 1998. ISSN 0163-5808. doi: http://doi.acm.org/10.1145/276305.276337.

[56] S. Chaudhuri and V. R. Narasayya. An efficient cost-driven index selection tool for microsoft sql server. In *VLDB*, pages 146–155, 1997. ISBN 1-55860-470-7.

[57] S. Chaudhuri, A. K. Gupta, and V. Narasayya. Compressing sql workloads. In *SIGMOD*, pages 488–499, 2002. ISBN 1-58113-497-5.

[58] Y. C. Cheng, L. Gruenwald, G. Ingels, and M. T. Thakkar. Evaluating partitioning techniques for main memory database: Horizontal and single vertical. In *ICCI*, pages 570–574, 1993. ISBN 0-8186-4212-2.

[59] J. Coleman and R. Grosman. Unlimited Scale-up of DB2 Using Server-assisted Client Redirect. http://ibm.co/fLR2cH, October 2005.

[60] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2): 1277–1288, Aug. 2008.

[61] G. Copeland, W. Alexander, E. Boughter, and T. Keller. Data placement in bubba. *SIGMOD*, 17(3): 99–108, 1988. ISSN 0163-5808. doi: http://doi.acm.org/10.1145/971701.50213.

[62] G. Copeland, M. Franklin, and G. Weikum. Uniform object management. volume 416 of *EDBT*, pages 253–268. 1990.

[63] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, M. S. Yasushi Saito, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's Globally-Distributed Database. In *OSDI*, 2012.

[64] J. Cowling and B. Liskov. Granola: low-overhead distributed transaction coordination. In *USENIX ATC*, pages 21–34, June 2012.

[65] P. Cudre-Mauroux, H. Kimura, K.-T. Lim, J. Rogers, R. Simakov, E. Soroush, P. Velikhov, D. L. Wang, M. Balazinska, J. Becla, D. DeWitt, B. Heath, D. Maier, S. Madden, J. Patel, M. Stonebraker, and S. Zdonik. A demonstration of SciDB: a science-oriented DBMS. *Proc. VLDB Endow.*, 2(2): 1534–1537, Aug. 2009.

[66] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *VLDB*, 3:48–57, 2010. ISSN 2150-8097.

[67] C. Curino, E. Jones, R. Popa, N. Malviya, E. Wu, S. Madden, H. Balakrishnan, and N. Zeldovich. Relational cloud: The case for a database service. In *CIDR*, pages 1–7, 2011.

[68] E. Danna and L. Perron. Structured vs. unstructured large neighborhood search: A case study on job-shop scheduling problems with earliness and tardiness costs. In *Principles and Practice of Constraint Programming*, volume 2833, pages 817–821, 2003.

[69] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41:205–220, October 2007. ISSN 0163-5980.

[70] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Commun. ACM*, 35(6):85–98, June 1992. ISSN 0001-0782.

[71] D. DeWitt, M. Smith, and H. Boral. A single-user performance evaluation of the teradata database machine. In *High Performance Transaction Systems*, volume 359 of *Lecture Notes in Computer Science*, pages 243–276. 1989.

[72] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. *SIGMOD Rec.*, 14(2):1–8, 1984.

[73] D. J. DeWitt, R. H. Gerber, G. Graefe, M. L. Heytens, K. B. Kumar, and M. Muralikrishna. GAMMA - a high performance dataflow database machine. In *VLDB*, pages 228–237, 1986. ISBN 0-934613-18-4.

[74] D. J. Dewitt, S. Ghandeharizadeh, and D. Schneider. A performance analysis of the gamma database machine. SIGMOD, pages 350–360, 1988.

[75] D. J. Dewitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen. The GAMMA database machine project. *IEEE Trans. on Knowl. and Data Eng.*, 2(1):44–62, Mar. 1990.

[76] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: Sql server's memory-optimized oltp engine. SIGMOD, pages 1243–1254, 2013.

[77] J. Dittrich and A. Jindal. Towards a one size fits all database architecture. In *CIDR*, pages 195–198, 2011.

[78] N. Du, X. Ye, and J. Wang. Towards workflow-driven database system workload modeling. In *DBTest '09*, pages 1–6, 2009. ISBN 978-1-60558-706-6. doi: http://doi.acm.org/10.1145/1594156.1594169.

[79] S. Duan, V. Thummala, and S. Babu. Tuning database configuration parameters with ituned. *VLDB*, 2:1246–1257, August 2009.

[80] S. S. Elnaffar. A methodology for auto-recognizing dbms workloads. In *CASCON*, page 2. IBM Press, 2002.

[81] R. Epstein, M. Stonebraker, and E. Wong. Distributed query processing in a relational data base system. SIGMOD, pages 169–180, 1978.

[82] R. Escriva, B. Wong, and E. G. Sirer. HyperDex: a distributed, searchable key-value store. *SIGCOMM Comput. Commun. Rev.*, 42(4):25–36, Aug. 2012.

[83] A. et al, D. Bitton, M. Brown, R. Catell, S. Ceri, T. Chou, D. DeWitt, D. Gawlick, H. Garcia-Molina, B. Good, J. Gray, P. Homan, B. Jolls, T. Lukes, E. Lazowska, J. Nauman, M. Pong, A. Spector, K. Trieber, H. Sammer, O. Serlin, M. Stonebraker, A. Reuter, and P. Weinberger. A measure of transaction processing power. *Datamation*, 31(7):112–118, Apr. 1985.

[84] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. SAP HANA database: data management for modern business applications. *SIGMOD Rec.*, 40(4):45–51, 2012. ISSN 0163-5808.

[85] B. Fitzpatrick. Distributed caching with Memcached. *Linux J.*, 2004(124):5–, Aug. 2004. ISSN 1075-3583.

[86] F. Focacci, F. Laburthe, and A. Lodi. *Handbook of Metaheuristics*, chapter Local Search and Constraint Programming. Springer, 2003.

[87] N. Folkman. So, that was a bummer. http://blog.foursquare.com/2010/10/05/so-that-was-a-bummer/, October 2010.

[88] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Trans. on Knowl. and Data Eng.*, 4(6):509–516, Dec. 1992.

[89] S. Ghandeharizadeh and D. J. DeWitt. Hybrid-range partitioning strategy: A new declustering strategy for multiprocessor database machines. VLDB, pages 481–492, 1990.

[90] S. Ghandeharizadeh, D. J. DeWitt, and W. Qureshi. A performance analysis of alternative multi-attribute declustering strategies. *SIGMOD*, 21(2):29–38, 1992. ISSN 0163-5808. doi: http://doi.acm.org/10.1145/141484.130293.

[91] S. Gilbert and N. A. Lynch. Perspectives on the CAP theorem. *Computer*, 45(2):30–36, 2012.

[92] G. Graefe. The five-minute rule twenty years later, and how flash memory changes the rules. DaMoN, pages 6:1–6:9, 2007.

[93] J. Gray. *Concurrency Control and Recovery in Database Systems*, chapter Notes on data base operating systems, pages 393–481. Springer-Verlag, 1978.

[94] J. Gray. The transaction concept: virtues and limitations. volume 7 of *VLDB*, pages 144–154, 1981.

[95] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. 1992. ISBN 1558601902.

[96] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*, chapter What is a Transaction Processing System?, pages 5–21. 1992. ISBN 1558601902.

[97] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. SIGMOD, pages 173–182, 1996.

[98] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Modelling in data base management systems. chapter Granularity of locks and degrees of consistency in a shared data base, pages 365–393. 1976.

[99] L. Gruenwald and M. H. Eich. Database partitioning techniques to support reload in a main memory database system: MARS. In *International Conference on Parallel Processing and Databases*, pages 107–109, March 1990.

[100] L. Gruenwald and M. H. Eich. Selecting a database partitioning technique. *Journal of Database Management*, 4(3):27–39, 1993.

[101] C. Gupta, A. Mehta, and U. Dayal. PQR: Predicting Query Execution Times for Autonomous Workload Management. In *ICAC*, pages 13–22, 2008. ISBN 978-0-7695-3175-5.

[102] H. Gupta, V. Harinarayan, A. Rajaraman, and J. D. Ullman. Index selection for olap. In *ICDE*, pages 208–219, 1997. ISBN 0-8186-7807-0.

[103] I. Guyon, S. Gunn, M. Nikravesh, and L. A. Zadeh. *Feature Extraction: Foundations and Applications*. Springer-Verlag, 2006. ISBN 3540354875.

[104] H-Store Project. AuctionMark: A Benchmark for High-Performance OLTP Systems. `http://hstore.cs.brown.edu/projects/auctionmark`, .

[105] H-Store Project. The SEATS Airline Ticketing Systems Benchmark. `http://hstore.cs.brown.edu/projects/seats`, .

[106] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, Dec. 1983.

[107] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The WEKA data mining software: an update. *SIGKDD Explorations Newsletter*, 11:10–18, November 2009. ISSN 1931-0145.

[108] M. Hammer and A. Chan. Index selection in a self-adaptive data base management system. In *SIGMOD*, pages 1–8, 1976.

[109] M. Hammer and B. Niamir. A heuristic approach to attribute partitioning. In *SIGMOD*, pages 93–101, 1979. ISBN 0-89791-001-X. doi: http://doi.acm.org/10.1145/582095.582110.

[110] M. Hammer and D. Shipman. Reliability mechanisms for SDD-1: a system for distributed databases. *ACM Trans. Database Syst.*, 5(4):431–466, Dec. 1980.

[111] J. R. Haritsa, K. Ramamritham, and R. Gupta. The prompt real-time commit protocol. *IEEE Trans. Parallel Distrib. Syst.*, 11:160–181, February 2000. ISSN 1045-9219.

[112] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. In *SIGMOD*, pages 981–992, 2008. ISBN 978-1-60558-102-6.

[113] P. Helland. Life beyond distributed transactions: an apostate's opinion. In *CIDR*, pages 132–141, 2007.

[114] P. Helland, H. Sammer, J. Lyon, R. Carr, P. Garrett, and A. Reuter. Group commit timers and high volume transaction systems. In *Proceedings of the 2nd International Workshop on High Performance Transaction Systems*, pages 301–329, 1989. ISBN 3-540-51085-0.

[115] J. M. Hellerstein and M. Stonebraker. Readings in database systems. chapter Transaction Management, pages 238–243. 4th edition, 1998.

[116] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, Jan. 1991.

[117] G. Herman, K. C. Lee, and A. Weinrib. The datacycle architecture for very high throughput database systems. SIGMOD, pages 97–103, 1987.

[118] M. Heytens, S. Listgarten, M.-A. Neimat, and K. Wilkinson. Smallbase: A main-memory dbms for high-performance applications. Technical report, Hewlett-Packard Laboratories, 1995.

[119] M. Holze and N. Ritter. Towards workload shift detection and prediction for autonomic databases. In *PIKM*, pages 109–116, 2007.

[120] M. Holze and N. Ritter. Autonomic Databases: Detection of Workload Shifts with n-Gram-Models. In *ADBIS*, pages 127–142, 2008. ISBN 978-3-540-85712-9.

[121] R. A. Howard. *Dynamic Programming and Markov Processes*. MIT˜Press, 1960.

[122] J. Hugg. New age transactional systems – not your grandpa's OLTP. StrangeLoop Conference, 2011.

[123] S.-O. Hvasshovd, O. Torbjørnsen, S. E. Bratsberg, and P. Holager. The ClustRa telecom database: High availability, high throughput, and real-time response. VLDB, pages 469–477, 1995.

[124] M. Y. L. Ip, L. V. Saxton, and V. V. Raghavan. On the selection of an optimal set of indexes. *IEEE Trans. Softw. Eng.*, 9(2):135–143, 1983. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/TSE.1983.236458.

[125] H. V. Jagadish, D. F. Lieuwen, R. Rastogi, A. Silberschatz, and S. Sudarshan. Dali: A high performance main memory storage manager. In *VLDB*, pages 48–59, 1994. ISBN 1-55860-153-8.

[126] R. Johnson, I. Pandis, and A. Ailamaki. Improving oltp scalability using speculative lock inheritance. *Proc. VLDB Endow.*, 2(1):479–489, Aug. 2009. ISSN 2150-8097.

[127] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: a scalable storage manager for the multicore era. EDBT, pages 24–35, 2009.

[128] R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki. Aether: a scalable approach to logging. *Proc. VLDB Endow.*, 3(1-2):681–692, 2010. ISSN 2150-8097.

[129] R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki. Scalability of write-ahead logging on multicore and multisocket hardware. *The VLDB Journal*, 21(2):239–263, Apr. 2012.

[130] E. P. Jones. *Fault-Tolerant Distributed Transactions for Partitioned OLTP Databases*. PhD thesis, MIT, 2011.

[131] E. P. Jones, D. J. Abadi, and S. Madden. Low overhead concurrency control for partitioned main memory databases. In *SIGMOD*, pages 603–614, 2010. ISBN 978-1-4503-0032-2.

[132] S. Kadioglu, Y. Malitsky, M. Sellmann, and K. Tierney. Isac – instance-specific algorithm configuration. In *ECAI*, pages 751–756, 2010.

[133] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. *Proc. VLDB Endow.*, 1(2):1496–1499, 2008.

[134] A. Kemper and T. Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. ICDE, pages 195–206, 2011.

[135] M. L. Kersten, P. M. Apers, M. A. Houtsma, E. J. Kuyk, and R. L. Weg. A distributed, main-memory database machine. In M. Kitsuregawa and H. Tanaka, editors, *Database Machines and Knowledge Base Machines*, volume 43 of *The Kluwer International Series in Engineering and Computer Science*, pages 353–369. 1988.

[136] H. Kolltveit and S.-O. Hvasshovd. The circular two-phase commit protocol. In *Proceedings of the 12th international conference on Database systems for advanced applications*, DASFAA'07, pages 249–261, 2007. ISBN 978-3-540-71702-7. URL `http://dl.acm.org/citation.cfm?id=1783823.1783854`.

[137] J. Krueger, C. Kim, M. Grund, N. Satish, D. Schwalb, J. Chhugani, H. Plattner, P. Dubey, and A. Zeier. Fast updates on Read-Optimized databases using Multi-Core CPUs. *VLDB*, 5:61–72, September 2011.

[138] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, June 1981. ISSN 0362-5915.

[139] T. Lahiri, M.-A. Neimat, and S. Folkman. Oracle TimesTen: An in-memory database for enterprise applications. *IEEE Data Eng. Bull.*, 36(2):6–13, 2013.

[140] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.

[141] L. Lamport. Paxos Made Simple. *SIGACT News*, (4):51–58, Dec. 2001.

[142] B. W. Lampson and D. B. Lomet. A new presumed commit optimization for two phase commit. In *Proceedings of the 19th International Conference on Very Large Data Bases*, VLDB '93, pages 630–640, 1993. ISBN 1-55860-152-X.

[143] P.-A. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory databases. *VLDB*, 5(4):298–309, Dec. 2011. ISSN 2150-8097.

[144] J. Lee, M. Muehle, N. May, F. Faerber, V. S. H. Plattner, J. Krueger, and M. Grund. High-performance transaction processing in SAP HANA. *IEEE Data Eng. Bull.*, 36(2):28–33, 2013.

[145] T. J. Lehman. *Design and performance evaluation of a main memory relational database system*. PhD thesis, University of Wisconsin–Madison, 1986.

[146] T. J. Lehman and M. J. Carey. A study of index structures for main memory database management systems. VLDB, pages 294–303, 1986.

[147] T. J. Lehman and M. J. Carey. Query processing in main memory database management systems. SIGMOD, pages 239–250, 1986.

[148] T. J. Lehman and M. J. Carey. A recovery algorithm for a high-performance memory-resident database system. SIGMOD, pages 104–117, 1987.

[149] T. J. Lehman, E. J. Shekita, and L.-F. Cabrera. An evaluation of starburst's memory resident storage component. *TKDE*, 4:555–566, 1992.

[150] E. Levy and A. Silberschatz. Incremental recovery in main memory database systems. *IEEE Trans. on Knowl. and Data Eng.*, 4(6):529–540, Dec. 1992.

[151] K. Li and J. F. Naughton. Multiprocessor main memory transaction processing. *DPDS*, pages 177–187, 1988.

[152] J. Lindstrom, V. Raatikka, J. Ruuth, P. Soini, and K. Vakkila. IBM solidDB: In-memory database optimized for extreme speed and availability. *IEEE Data Eng. Bull.*, 36(2):14–20, 2013.

[153] M. Livny, S. Khoshafian, and H. Boral. Multi-disk management algorithms. *SIGMETRICS Perform. Eval. Rev.*, 15(1):69–77, 1987. ISSN 0163-5999. doi: http://doi.acm.org/10.1145/29904.29914.

[154] L. F. Mackert and G. M. Lohman. R* optimizer validation and performance evaluation for local queries. SIGMOD, pages 84–95, 1986.

[155] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. Recovery algorithms for in-memory OLTP databases. *In Submission*, 2013.

[156] S. Manegold. *Understanding, Modeling, and Improving Main-Memory Database Performance*. PhD thesis, Universiteit van Amsterdam, December 2002.

[157] S. Manegold, P. Boncz, and M. L. Kersten. Generic database cost models for hierarchical memory systems. In *VLDB*, pages 191–202, 2002.

[158] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. EuroSys, pages 183–196, 2012.

[159] M. Mehta and D. J. DeWitt. Data placement in shared-nothing parallel database systems. *The VLDB Journal*, 6(1):53–72, 1997. ISSN 1066-8888. doi: http://dx.doi.org/10.1007/s007780050033.

[160] C. Mohan, B. Lindsay, and R. Obermarck. Transaction management in the r* distributed database management system. *ACM Trans. Database Syst.*, 11(4):378–396, Dec. 1986. ISSN 0362-5915.

[161] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, 1992. ISSN 0362-5915.

[162] C. Monash. H-Store is now VoltDB. `http://www.dbms2.com/2009/06/22/h-store-horizontica-voltdb/`, June 2009.

[163] T. Mühlbauer, W. Rödiger, A. Reiser, A. Kemper, and T. Neumann. ScyPer: elastic olap throughput on transactional data. DanaC, pages 11–15, 2013.

[164] R. Mukkamala, S. C. Bruell, and R. K. Shultz. Design of partially replicated distributed database systems: an integrated methodology. *SIGMETRICS Perform. Eval. Rev.*, 16(1):187–196, 1988. ISSN 0163-5999. doi: http://doi.acm.org/10.1145/1007771.55617.

[165] O. Mutlu, H. Kim, and Y. N. Patt. Techniques for efficient processing in runahead execution engines. ISCA, pages 370–381, 2005. ISBN 0-7695-2270-X.

[166] R. Nehme and N. Bruno. Automated partitioning design in parallel database systems. In *SIGMOD*, SIGMOD, pages 1137–1148, 2011. ISBN 978-1-4503-0661-4.

[167] E. B. Nightingale, P. M. Chen, and J. Flinn. Speculative execution in a distributed file system. *ACM Trans. Comput. Syst.*, 24(4):361–392, Nov. 2006. ISSN 0734-2071.

[168] C. Nikolaou, A. Labrinidis, V. Bohn, D. Ferguson, M. Artavanis, C. Kloukinas, and M. Marazakis. The impact of workload clustering on transaction routing. Technical report, FORTH-ICS TR-238, 1998.

[169] C. N. Nikolaou, M. Marazakis, and G. Georgiannakis. Transaction routing for distributed OLTP systems: survey and recent results. *Inf. Sci.*, 97:45–82, 1997. ISSN 0020-0255.

[170] *NuoDB Emergent Architecture – A 21st Century Transactional Relational Database Founded On Partial, On-Demand Replication*. NuoDB LLC., Jan. 2013.

[171] R. Obermarck. Distributed deadlock detection algorithm. *ACM Trans. Database Syst.*, 7(2):187–208, June 1982.

[172] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Inf.*, 33(4):351–385, June 1996.

[173] P. E. O'Neil. The escrow transactional method. *ACM Trans. Database Syst.*, 11:405–430, December 1986. ISSN 0362-5915.

[174] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for ramclouds: scalable high-performance storage entirely in dram. *SIGOPS Oper. Syst. Rev.*, 43(4): 92–105, Jan. 2010. ISSN 0163-5980.

[175] M. T. Ozsu. *Principles of Distributed Database Systems*. 3rd edition, 2007. ISBN 9780130412126.

[176] S. Padmanabhan. *Data placement in shared-nothing parallel database systems*. PhD thesis, University of Michigan, 1992.

[177] I. Pandis. *Scalable Transaction Processing through Data-Oriented Execution*. PhD thesis, Carnegie Mellon, 2012.

[178] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. *Proc. VLDB Endow.*, 3:928–939, September 2010.

[179] S. Papadomanolakis and A. Ailamaki. Autopart: Automating schema design for large scientific databases using data partitioning. In *SSDBM*, 2004.

[180] A. Pavlo, E. P. Jones, and S. Zdonik. On predictive modeling for optimizing transaction execution in parallel oltp systems. *Proc. VLDB Endow.*, 5:85–96, October 2011.

[181] A. Pavlo, C. Curino, and S. B. Zdonik. Skew-Aware Automatic Data Partitioning in Shared-Nothing, Parallel OLTP Systems, 2012.

[182] S. Pilarski and T. Kameda. Checkpointing for distributed databases: Starting from the basics. *IEEE Trans. Parallel Distrib. Syst.*, 3(5):602–610, Sept. 1992.

[183] D. Porobic, I. Pandis, M. Branco, P. TÃűzÃijn, and A. Ailamaki. OLTP on Hardware Islands. *Proc. VLDB Endow.*, 5:1447–1458, July 2012.

[184] E. Rahm. A framework for workload allocation in distributed transaction processing systems. *J. Syst. Softw.*, 18:171–190, May 1992. ISSN 0164-1212.

[185] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. 3rd edition, 2003.

[186] P. Ranganathan. From microprocessors to nanostores: Rethinking data-centric systems. *Computer*, 44 (1):39–48, Jan. 2011. ISSN 0018-9162.

[187] P. Ranganathan, K. Gharachorloo, S. V. Adve, and L. A. Barroso. Performance of database workloads on shared-memory systems with out-of-order processors. *SIGPLAN Not.*, 33(11):307–318, Oct. 1998. ISSN 0362-1340.

[188] J. Rao, C. Zhang, N. Megiddo, and G. Lohman. Automating physical database design in a parallel database. In *SIGMOD*, pages 558–569, 2002. ISBN 1-58113-497-5. doi: http://doi.acm.org/10.1145/564691.564757.

[189] P. K. Reddy and M. Kitsuregawa. Speculative Locking Protocols to Improve Performance for Distributed Database Systems. *IEEE Trans. on Knowl. and Data Eng.*, 16(2):154–169, 2004.

[190] D. Roberts, J. Chang, P. Ranganathan, and T. N. Mudge. Is storage hierarchy dead? co-located compute-storage nvram-based architectures for data-centric workloads. Technical Report HPL-2010-119, HP Labs, 2010.

[191] J. B. Rothnie, Jr., P. A. Bernstein, S. Fox, N. Goodman, M. Hammer, T. A. Landers, C. Reeve, D. W. Shipman, and E. Wong. Introduction to a system for distributed databases (SDD-1). *ACM Trans. Database Syst.*, 5(1):1–17, Mar. 1980.

[192] K. Salem and H. Garcia-Molina. System M: A transaction processing testbed for memory resident data. *IEEE Trans. on Knowl. and Data Eng.*, 2(1):161–172, Mar. 1990.

[193] G. Samaras, K. Britton, A. Citron, and C. Mohan. Two-phase commit optimizations and tradeoffs in the commercial environment. In *Proceedings of the Ninth International Conference on Data Engineering*, pages 520–529, 1993. ISBN 0-8186-3570-3.

[194] C. Sapia. PROMISE: Predicting Query Behavior to Enable Predictive Caching Strategies for OLAP Systems. In *DaWaK*, pages 224–233, 2000. ISBN 3-540-67980-4.

[195] P. Scheuermann, G. Weikum, and P. Zabback. Data partitioning and load balancing in parallel disk systems. *The VLDB Journal*, 7(1):48–66, 1998. ISSN 1066-8888. doi: http://dx.doi.org/10.1007/s007780050053.

[196] S. A. Schuster. Relational data base management for on-line transaction processing. Technical report, Tandem, Feb. 1981.

[197] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, pages 23–34, 1979. ISBN 0-89791-001-X. doi: http://doi.acm.org/10.1145/582095.582099.

[198] R. Shoup and D. Pritchett. The ebay architecture. SD Forum, November 2006.

[199] V. Sikka, F. Färber, W. Lehner, S. K. Cha, T. Peh, and C. Bornhövd. Efficient transaction processing in SAP HANA database: the end of a column store myth. SIGMOD, pages 731–742, 2012.

[200] A. J. Smith. Sequentiality and prefetching in database systems. *ACM Trans. Database Syst.*, 3(3): 223–247, Sept. 1978. ISSN 0362-5915.

[201] J. Sobel. Scaling Out (Facebook). `http://on.fb.me/p7i7eK`, April 2006.

[202] G. Soundararajan, M. Mihailescu, and C. Amza. Context-aware prefetching at the storage server. In *USENIX ATC*, pages 377–390, 2008. URL `http://dl.acm.org/citation.cfm?id=1404014.1404045`.

[203] R. Stoica and A. Ailamaki. Enabling efficient os paging for main-memory OLTP databases. In *DaMon*, 2013.

[204] M. Stonebraker. Concurrency control and consistency of multiple copies of data in distributed INGRES. *IEEE Trans. Softw. Eng.*, 5(3):188–194, May 1979.

[205] M. Stonebraker. MUFFIN: a distributed data base machine. Technical report, University of California, Berkeley. Electronics Research Laboratory, 1979.

[206] M. Stonebraker. The Case for Shared Nothing. *Database Engineering*, 9:4–9, 1986.

[207] M. Stonebraker and R. Cattell. 10 rules for scalable performance in 'simple operation' datastores. *Commun. ACM*, 54:72–80, June 2011.

[208] M. Stonebraker and L. A. Rowe. The design of POSTGRES. SIGMOD, pages 340–355, 1986.

[209] M. Stonebraker, G. Held, E. Wong, and P. Kreps. The design and implementation of INGRES. *ACM Trans. Database Syst.*, 1(3):189–222, Sept. 1976.

[210] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-Store: A column-oriented dbms. In *VLDB*, pages 553–564, 2005.

[211] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *VLDB*, pages 1150–1160, 2007. ISBN 978-1-59593-649-3.

[212] Tandem Database Group. NonStop SQL, A Distributed, High-Performance, High-Availability Implementation of SQL. Technical report, Tandem, Apr. 1987.

[213] A. L. Tatarowicz, C. Curino, E. P. C. Jones, and S. Madden. Lookup tables: Fine-grained partitioning for distributed databases. ICDE, pages 102–113, 2012.

[214] F. Tauheed, T. Heinis, F. Schürmann, H. Markram, and A. Ailamaki. SCOUT: prefetching for latent structure following queries. *VLDB*, 5(11):1531–1542, 2012.

[215] The Transaction Processing Council. TPC-E Benchmark (Draft Revision 0.32.2g). `http://www.tpc.org/tpce/`, July 2006.

[216] The Transaction Processing Council. TPC-C Benchmark (Revision 5.9.0). `http://www.tpc.org/tpcc/spec/tpcc_current.pdf`, June 2007.

[217] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst.*, 4(2):180–209, June 1979.

[218] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD*, pages 1–12, 2012. ISBN 978-1-4503-1247-9.

[219] TimesTen Team. In-memory data management for consumer transactions the timesten approach. SIGMOD, pages 528–529, 1999. ISBN 1-58113-084-8.

[220] P. Tözün, I. Pandis, C. Kaynak, D. Jevdjic, and A. Ailamaki. From A to E: analyzing TPC's OLTP benchmarks: the obsolete, the ubiquitous, the unexplored. EDBT, pages 17–28, 2013.

[221] G. Valentin, M. Zuliani, D. Zilio, G. Lohman, and A. Skelley. DB2 advisor: an optimizer smart enough to recommend its own indexes. In *ICDE*, pages 101–110, 2000.

[222] W. Vogels. Eventually consistent. *Queue*, 6:14–19, October 2008.

[223] C. B. Walton, A. G. Dale, and R. M. Jenevein. A taxonomy and performance model of data skew effects in parallel joins. In *VLDB '91*, pages 537–548, 1991. ISBN 1-55860-150-3.

[224] A. Whitney, D. Shasha, and S. Apter. High Volume Transaction Processing Without Concurrency Control, Two Phase Commit, SQL or C++. In *HPTS*, 1997.

[225] R. Williams, D. Daniels, L. Haas, G. Lapis, B. Lindsay, P. Ng, R. Obermarck, P. Selinger, A. Walker, P. Wilms, and R. Yost. Distributed systems, vol. ii: distributed data base systems. chapter R*: an overview of the architecture, pages 435–461. 1986.

[226] A. Wolski. TATP Benchmark Description (Version 1.0). `http://tatpbenchmark.sourceforge.net`, March 2009.

[227] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Satzilla: portfolio-based algorithm selection for sat. *J. Artif. Int. Res.*, 32:565–606, June 2008. ISSN 1076-9757.

[228] Q. Yao, A. An, and X. Huang. Mining and modeling database user access patterns. In *Foundations of Intelligent Systems*, volume 4203 of *Lecture Notes in Computer Science*, pages 493–503. 2006.

[229] P. S. Yu, M.-S. Chen, H.-U. Heiss, and S. Lee. On workload characterization of relational database environments. *IEEE Trans. Softw. Eng.*, 18(4):347–355, 1992. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/32.129222.

[230] D. C. Zilio. *Physical Database Design Decision Algorithms and Concurrent Reorganization for Parallel Database Systems*. PhD thesis, University of Toronto, 1998.

[231] D. C. Zilio, A. Jhingran, and S. Padmanabhan. Partitioning key selection for shared-nothing parallel database system. Technical Report 87739, IBM Research, November 1994.

[232] D. C. Zilio, J. Rao, S. Lightstone, G. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden. DB2 design advisor: integrated automatic physical database design. In *VLDB*, pages 1087–1097, 2004. ISBN 0-12-088469-0.