Adaptive Data Storage and Placement in Distributed Database Systems

by

Michael Abebe

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2022

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	Aaron Elmore Associate Professor, Dept. of Computer Science University of Chicago
Supervisor:	Khuzaima Daudjee Associate Professor, School of Computer Science University of Waterloo
Internal Members:	M. Tamer Özsu University Professor, School of Computer Science University of Waterloo
	Samer Al-Kiswany Associate Professor, School of Computer Science University of Waterloo
Internal-External Member:	Paul Ward Associate Professor, Dept. of Electrical and Computer Engineering University of Waterloo

Author's Declaration

Statement of Contributions

Some portions of this thesis are based on peer-reviewed joint work with Khuzaima Daudjee, Brad Glasbergen and Horatiu Lazu, in which I am the first author and the primary contributor [12, 13, 14, 15].

Abstract

Distributed database systems are widely used to provide scalable storage, update and query facilities for application data. Distributed databases primarily use data replication and data partitioning to spread load across nodes or sites. The presence of hotspots in workloads, however, can result in imbalanced load on the distributed system resulting in performance degradation. Moreover, updates to partitioned and replicated data can require expensive distributed coordination to ensure that they are applied atomically and consistently. Additionally, data storage formats, such as row and columnar layouts, can significantly impact latencies of mixed transactional and analytical workloads. Consequently, how and where data is stored among the sites in a distributed database can significantly affect system performance, particularly if the workload is not known ahead of time. To address these concerns, this thesis proposes adaptive data placement and storage techniques for distributed database systems.

This thesis demonstrates that the performance of distributed database systems can be improved by automatically adapting how and where data is stored by leveraging online workload information. A two-tiered architecture for adaptive distributed database systems is proposed that includes an adaptation advisor that decides at which site(s) and how transactions execute. The adaptation advisor makes these decisions based on submitted transactions. This design is used in three adaptive distributed database systems presented in this thesis: (i) DynaMast that efficiently transfers data mastership to guarantee single-site transactions while maintaining well-understood and established transactional semantics, (ii) MorphoSys that selectively and adaptively replicates, partitions and remasters data based on a learned cost model to improve transaction processing, and (iii) Proteus that uses learned workload models to predictively and adaptively change storage layouts to support both high transactional throughput and low latency analytical queries.

Collectively, this thesis is a concrete step towards autonomous database systems that allow users to specify only the data to store and the queries to execute, leaving the system to judiciously choose the storage and execution mechanisms to deliver high performance.

Acknowledgements

This thesis would not be possible without the help of so many people in my life. It would be impossible to acknowledge everyone, but I will try my best.

I thank my advisor Professor Khuzaima Daudjee for the (seemingly endless) amount of time he has spent with me working on research. Khuzaima's unique ability to recall related work from all areas of computer science broadened the depth of my knowledge. In our meetings, Khuzaima helped me see the bigger picture of the technical problem I was trying to solve, ensuring that my work remained focused. Finally, through his diligent reading, comments, and examination of the meaning of every word, sentence, paragraph and section, Khuzaima has significantly improved my writing. I am thankful to have Khuzaima as an advisor.

My committee members, Professors Tamer Özsu, Samer Al-Kiswany, Paul Ward, and Aaron Elmore, provided valuable comments and asked interesting questions that helped refine this thesis. I sincerely appreciate the time they spent reading and engaging with my work. Additionally, Professors Özsu and Al-Kiswany, provided helpful comments on my thesis proposal that influenced the structure of this thesis. Furthermore, the work in Chapter 4 of this thesis began as a course project in Professor Özsu's graduate database course and benefitted from his feedback at that time.

I was lucky to start grad school a few months after Brad Glasbergen and quickly become office-mates. It is not an understatement to say that many of the ideas in this thesis came following a long discussion between the two of us, often arguing over minute details at a whiteboard. Brad made late nights working on papers and long days debugging race conditions better. Navigating the challenges of being a grad student were eased by having Brad to talk to. I am thankful to have met, collaborated with and become close friends with Brad.

Throughout my time in grad school, I was lucky to collaborate with several people. Formally these collaborations are represented as co-authors on papers. I learned something from each of my co-authors (Khuzaima Daudjee, Brad Glasbergen, Horatiu Lazu, Yuangfeng Tian, Anil Paçaci, Amit Levi, Kyle Langendoen, Scott Foggo, Jian Zhao and Daniel Vogel). In particular, I wish to thank Horatiu and Yuangfeng, two excellent undergraduate research assistants, for contributing ideas and implementing code on my research projects. In this thesis, Horatiu was key in the development of the different learning models used in Chapter 5. Informally, the faculty and student members of the Data Systems Group (DSG) at the University of Waterloo during my time as a student helped provide feedback on my research during seminar presentations. The broader database community also provided invaluable feedback on the work contained in this thesis, both through detailed reviews and discussions at conferences.

I wish to acknowledge three people who helped spark my interest in computer science, research, and distributed systems: Kevin Wells, Steven Zimmerly, and Henry Filgueiras. Mr. Wells encouraged me to learn to program on my TI-82 calculator in high school math class, introducing me to the world of computer science. Professor Zimmerly taught me the fundamentals of conducting research, ranging from managing experimental data to testing a hypothesis, all in the context of mobile RNA. During my undergrad, Henry hired me to work on distributed systems at the core of Facebook, when my only "systems" experience was taking an Operating Systems class. Working with Henry on these systems gave me a crash course in distributed systems, and I realized where my interests lie. The lessons Henry taught me will always stick with me, as he is the epitome of a "night watchman" [132].

My time as a grad student would have been a lot less enjoyable without the friendship of other grad students. The Davis Centre (DC) second-floor hallway was made much more lively (and frequently louder) because of the conversations between office 2118 (Brad, Nathan Harms and myself) and 2115 (Amit and Anil). Working across the hall from each other and across various research topics always prompted interesting discussions, which often veered to life more broadly. Frequent walks as a whole, or as subsets, to the Math C&D to get donuts or other pastries were often the remedy to whatever obstacles we faced in our research that day.

More generally, I am grateful to my many friends who have been there for me. In particular, Dan Fang and Andrei Curelea, always made time to catch up with me whenever they were in Waterloo or I was in the Bay Area. Brad and Becca Mayers formed a "bubble" with myself and Kathleen during the early days of the COVID-19 pandemic; our many meals and dog walks ensured we did not go crazy at-home. Finally, Farzaana Coovadia and Colin Wong have not just been great friends to me, but importantly, made sure that Kathleen was never alone when I was travelling or busy with a paper.

My family has been a constant source of support for me throughout the years. My parents, Nancy and Gashaw, emphasized the importance of education from a young age and have always supported mine. Weekend trips to Annan to visit them, my grandparents, aunts and uncles were a welcome break from my studies. My great-aunt and uncle, Herb and Vivian, greeted me in Waterloo when I first moved here in 2011 and always welcomed me into their home. The (extended) Read family (George, Fay, Allan, Robert, Emma and Jenna) were always curious about and supportive of my work. Holiday trips to Maple

Ridge to visit and relax with them were a refreshing break from research.

Completing this thesis without our dog, Rosie would have been much less pleasant. Rosie entered our lives in May 2020, early in the COVID-19 pandemic, when I began working on Chapter 5 of this thesis. As my office in DC was replaced with my at-home office, going for walks with Rosie ensured that I took breaks from my computer, which were essential in clearing my mind.

Finally and most importantly, I thank my wife, Kathleen Read, for her never-ending support and confidence in me completing this thesis. Whether I was tired from reading related work, singularly focused on paper deadlines or lost in thought thinking about transactions, Kathleen was always there. For this love and support, I will be forever thankful. Despite my name being on the front page of this thesis, I firmly believe that it is as much Kathleen's as it is mine.

Table of Contents

List of Figures

List of Tables

"The pessimist complains about the wind; the optimist expects it to change; the realist adjusts the sails."

— William Arthur Ward

"We can only see a short distance ahead, but we can see plenty there that needs to be done."

— Alan Turing

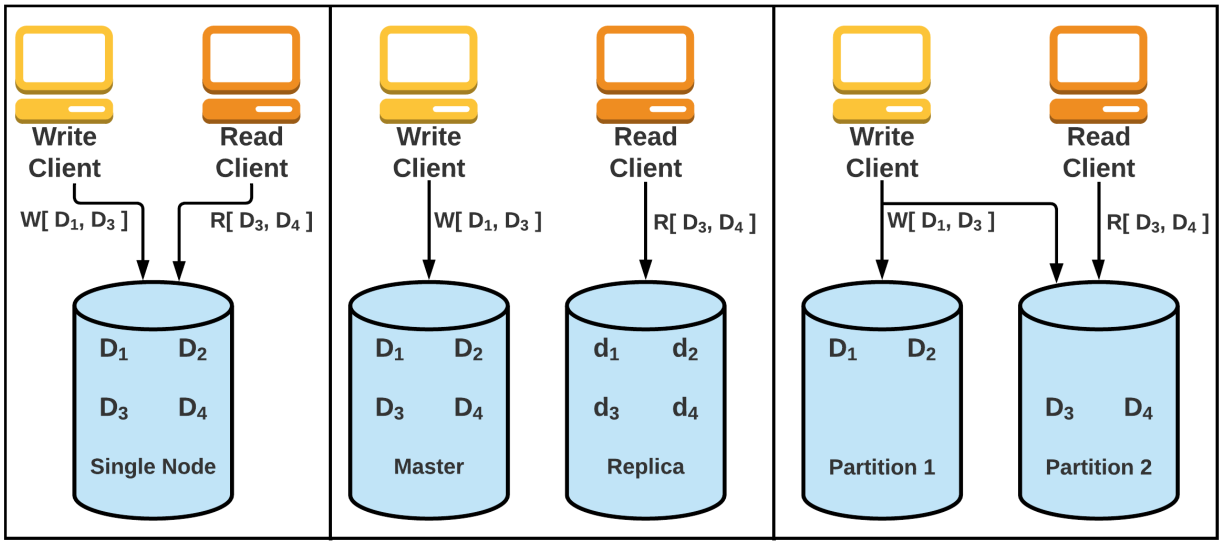
Chapter 1

Introduction

Data storage and management is at the core of nearly every software application. Database systems provide these features by presenting an interface to efficiently store and query data. Recently, the continuous growth in data processing demands has led to a renewed focus on large-scale distributed database management systems (DBMSs) [183, 122, 17, 145]. For example, consider an e-commerce organization that must process new online orders and analyze these orders for trends, such as the effects of a promotional sale. It is now common for such organizations to receive hundreds of thousands of order requests per second from clients, which places the underlying database under high load [38, 50]. These stringent demands exceed the capabilities of a single machine and therefore require a DBMS to scale and operate in a distributed environment.

Data replication and data partitioning are the two primary means of scaling out a DBMS, which I show in Figure 1.1, in contrast to a single node DBMS storing four data items (Figure 1.1a). Both data replication and partitioning distribute data over multiple database machines (nodes, or sites) to spread load. Replication creates copies of data and spreads them across nodes (Figure 1.1b). A client may access any node hosting its desired data to perform computations and retrieve results, which increases the amount of computing power available and can bring data closer to clients. Data partitioning (Figure 1.1c) spreads load across nodes by splitting data among the nodes. Partitioning enables a request to be processed by any node that holds the requested data and increases the amount of storage and compute power available to a request.

Although data replication and partitioning can improve the performance of data systems, these techniques also pose several challenges. Administrators must decide *where* and *how* to store data among the nodes in the distributed DBMS to support their workload



(a) Single Node DBMS (b) Data Replication (c) Data Partitioning

Figure 1.1: A database with four data items (D_1,...D_4), and examples of distributed data replication (1.1b) and data partitioning (1.1c).

demands. Choosing where and how to store data is generally a difficult problem, as each choice presents trade-offs, and a poor choice for the workload leads to low levels of system performance [53].

Partitioning data can result in contention, and skewed load on a node, that arises from high frequency data access concentrated on a subset of data or a hotspot. Consequently, rather than evenly distributing load among nodes, hotspots can result in the load being concentrated on a single database node. Furthermore, when data updates span multiple nodes, expensive distributed coordination protocols are required to ensure that the updates are applied atomically and consistently. Replication, while beneficial for bringing data closer to clients, consumes additional storage space. Moreover, replicated data must be maintained to ensure data consistency, which costs compute resources. Thus, the cost to store and manage replicated data must be carefully managed to ensure resources are being used efficiently.

Next, I examine the above trade-offs in more detail.

Partitioned DBMSs such as H-Store [96], and VoltDB [187] divide data among the nodes in the system, such that each data item is stored on one node (Figure 1.1c). This data partitioning allows transactions to execute at each site, limiting access to only the data stored there. Consequently, these partitioned DBMSs spread both the update and read load over multiple sites in the distributed system, but, the presence of hotspots or skewed access in the workload can result in uneven load distribution. Therefore, carefully partitioning where the data is stored, and thus partitioning the workload among the sites in the distributed system, is necessary to distribute the load evenly among the nodes. A performance concern with transactions that access, and in particular update, data at multiple sites is that they require costly synchronization and coordination across the sites at which they execute to guarantee transactional atomicity and consistency [94, 113, 76]. A popular coordination protocol used by both H-Store and VoltDB is two-phase commit, which significantly degrades the performance and scalability of distributed DBMSs due to multiple rounds of messages and blocking [94, 114, 76, 39].

Replicated DBMSs such as Amazon Aurora [196] and Postgres-R [205] maintain data copies at multiple sites (Figure 1.1b). Replication allows transactions to execute on copies of data. A key challenge with data replication is that updates to one copy of the data causes other copies to become stale [70]. Moreover, replicas risk being inconsistent in the face of multiple updates across the distributed system. Synchronous or eager update application at replicas are expensive due to the duplication of work at replicated sites during the critical path of the database operation. Consequently, lazy, or asynchronous, replica maintenance is used, which defers replica maintenance to outside transaction boundaries. Aurora and other replicated DBMSs [205, 196, 6, 99, 38, 5] adopt the lazy master replication model in which update transactions execute at a single node called the master site that holds all updateable or master copies of data. The remaining nodes in the distributed system hold read-only data replicas that are synchronized lazily. By placing the master copies of all data items at a single master site, all update transactions can execute and commit locally at that site. Similarly, read-only transactions can execute and commit locally at any replica. Thus, the single master architecture guarantees single-site transaction execution for all transactions, eliminating the need for expensive distributed transaction coordination. However, as the update workload scales up, the performance of the replicated system suffers as the single master site that executes all update transactions becomes a bottleneck [107, 36, 150].

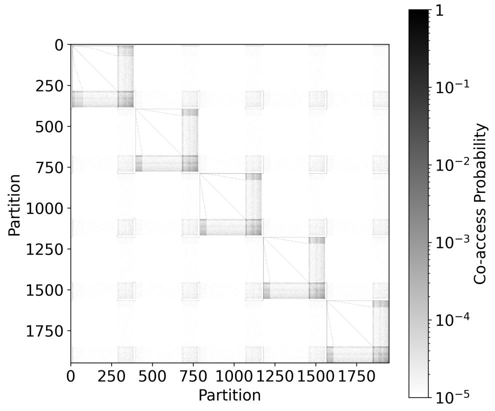
Finally, deciding how to store data at a site also poses a critical challenge. Online-transaction processing (OLTP) systems such as VoltDB [187] and X-Engine [85] store data tuples contiguously as *rows*, using the n-ary storage model [96, 33, 186]. Row-oriented storage is optimized for OLTP workloads that operate on a single record at a time and access many attributes. However, row formats result in poor support for analytical workloads that access many tuples at a time but only a subset of tuple attributes, as entire rows of data must be processed instead of only the relevant attributes. Thus, online-analytical processing (OLAP) systems such as MonetDB [34] and C-Store [182] store tuples attribute-at-a-time in *columns* using the decomposition storage model [9, 49, 34]. Although a column format works well for analytics, it is inefficient for OLTP workloads that update multiple columns within a transaction as each stored column is affected. Thus, neither storage format is efficient for *hybrid* workloads that span transaction and analytical processing.

These examples demonstrate that deciding how and where to store data in a distributed system is challenging, as each choice presents trade-offs. Offline tools [53, 156, 155, 176, 209, 75, 34, 158] help administrators decide where and how to store data, but generate *offline, static* decisions. These static decisions are not suitable if extensive information about the workload, such as the set of transactions that the DBMS will execute, is not available *a priori*. Additionally, these tools take time to execute and use. Consider the offline Schism [53] tool that can be used to select a static master placement scheme for distributed data in a DBMS. This tool requires collecting a representative trace of the workload, sampling transactions and data item accesses from this trace, constructing an access graph, running the graph partitioning algorithm Metis [98] over the graph, and then performing an explanation pass over the partitioning to generalize to data items that were not sampled. For example, running the Schism tool on a 1% sample of a 10-minute TPC-C workload trace takes 7 minutes to execute (generating the access graph takes 4 minutes, and partitioning the graph takes 3 minutes). This significant running time makes these tools undesirable for use in an online setting.

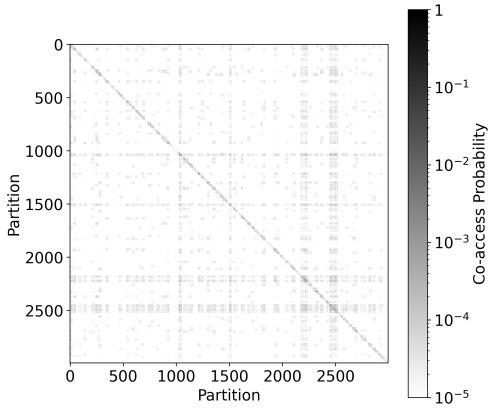
1.1 Workload Characteristics and Patterns

I now examine the characteristics of workloads that make them time-consuming and complex for administrators to understand. However, these properties typically result in workload patterns that a system can observe and utilize to make decisions effectively.

As previously mentioned, workloads can have hotspots. For example, consider Shopify's e-commerce database workload [143]. Users place orders for items; hence popular items in the real world will have more updates to their underlying database representation than unpopular items, which results in hotspots and contention on these data items in the system [143, 85]. Moreover, these hotspots can shift over time as the popularity of items changes or as a result of follow-the-sun cycles from geo-distribution of clients [191] resulting in access patterns with temporal locality. The DBMS can also suffer from short-term load spikes that occur because of a short burst of increased demand, such as during popular windows of online shopping [190, 85]. For example, the request rate can increase between 10 and 100 times on days like Black Friday, Cyber Monday, or Single's Day [85, 143]. Database workloads can also change over time as recent data becomes historical and less likely to be updated [26].



(a) TPC-C



(b) Twitter

Figure 1.2: Measured Co-Access Probabilities for the TPC-C and Twitter workloads.

There is also spatial locality in how data is accessed: shoppers may frequently place an order for groups of items together (e.g., wrapping paper and birthday cards), resulting in data co-access patterns. Consider Figure 1.2 which measures the spatial locality of transactions within two well known database benchmark workloads: TPC-C (Figure 1.2a) and Twitter (Figure 1.2b). Specifically, in Figure 1.2, on the x- and y-axis, I show data items grouped by partition and measure the percentage of transactions that co-access both partitions on the x and y-axis. As shown in Figure 1.2 both TPC-C and Twitter exhibit locality in how data is accessed. The locality in TPC-C arises from the schema structure, which is a tree-based schema derived from the warehouse and district structure. In this case, there are 5 warehouses with regular access patterns following the tree-based schema. However, there are still regions of data with seemingly less structured access patterns within a warehouse, such as on the stock table. The locality pattern in Twitter is not as easily interpretable as the access pattern arises from the relationships among users and tweets in a many-to-many relationship. Consequently, administrators may not fully understand or discern data co-access patterns ahead of time in both workloads, whereas a system can learn these patterns on the fly.

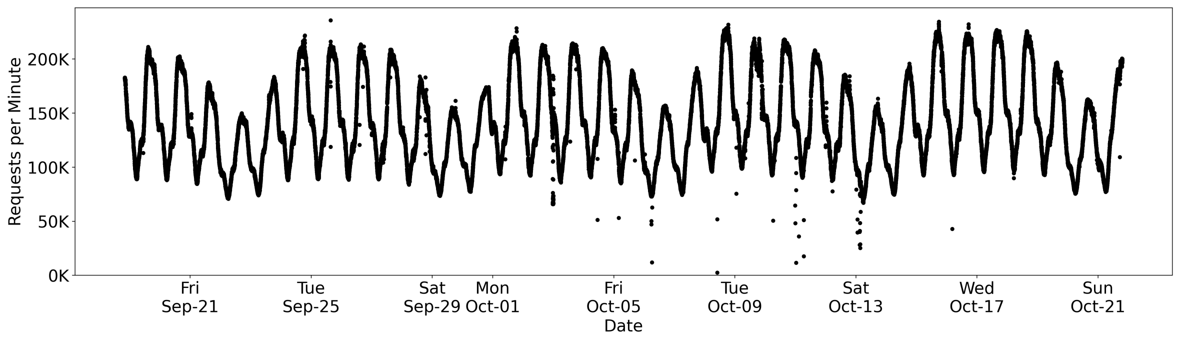


Figure 1.3: Wikipedia Access Frequency per Minute Over a Month

Figure 1.2 presented a summary of data locality statistics that were collected over a 20 minute period. However, workload patterns may change over time, as workloads are not static. These changes occur because of (i) changes in end-user behaviour resulting in changes in access frequency, (ii) short term load spikes because of increased demand, (iii) changes in access pattern types as data ages and becomes historical, and (iv) change over time due to follow the sun cycles. This is evident in Figure 1.3, which shows temporal access patterns arising from access frequency to Wikipedia [195] over a month. Hence, decisions made based on the workload at one time may yield poor performance as the workload evolves and changes.

1.2 Designing Adaptive Distributed DBMS

The prior observations about workload characteristics and the performance of distributed DBMS that rely on static data placement and storage decisions motivate systems that can dynamically change these decisions on the fly based on the workload. That is, if the system can *adapt*, or alter, how they store or place data, then such *adaptive* systems can provide several opportunities for performance improvements.

As discussed, workloads are not always static nor understood ahead of time. Adapting data storage and placement based on the workload at hand ensures that the system can continually operate at a desirable level of performance. Additionally, by making data storage and placement decisions adaptively, the system can converge to a set of decisions that a system administrator may not normally consider. For example, the prior discussion has contrasted the partitioned and replicated architectures, which differ in their replication choices, i.e., none or complete, respectively. However, an adaptive system can bridge the gap between the two architectures, making granular replication decisions for each data item: none, partial or complete.

There are several other challenges to building an adaptive distributed DBMS. The system must execute transactions efficiently while it adapts. It is not practical for the system to stop processing requests while changing how or where data is stored. Adaptive systems must perform their changes efficiently; if it takes minutes or hours to complete a change, then that change may no longer be beneficial to the system. Additionally, an adaptive system should not require prior workload knowledge, that is, information about the set of transactions that will execute and when, but should adapt based on the workload it observes as it executes. That is, the system's knowledge of the workload should be based on the transactions executed by the system at any point in time. By contrast, requiring prior knowledge of the workload restricts the set of workloads that the system can support. Finally, adaptive systems must prudently select beneficial changes and avoid unnecessary changes that they will undo in the immediate future. If changes are repeatedly made and undone, then the system wastes resources in making these changes.

1.3 Contributions

The prior discussion highlights the challenges of statically selecting where and how to store data in a distributed DBMS, along with the potential benefits of making these decisions adaptively. In this thesis, I demonstrate that the performance of distributed database management systems can be improved by automatically adapting how and where data is stored using online workload information.

To this end, I design and develop three adaptive distributed DBMSs, each addressing key research issues that I discussed above. Collectively these systems demonstrate the benefit of adaptive data management. Next, I will motivate the three systems and the research problems each address that allow them to adapt data storage and placement in a distributed setting.

1.3.1 Adaptive Dynamic Mastering for Replicated Systems

Recall that the single-master architecture guarantees that transactions execute at a single site, thereby avoiding expensive distributed coordination. However, the single-master becomes a bottleneck as the update workload scales up. In contrast, the multi-master architecture, distributes master copies of data, and therefore updates, to multiple sites [184].

However, the distribution of master copies results in transactions updating data at multiple sites [53, 76, 94] which require expensive distributed commit protocols, such as two-phase commit, to ensure transactional consistency and atomicity for multi-site update transactions, multi-master systems must employ a distributed commit protocol, e.g., two-phase commit. Unfortunately, such commit protocols significantly degrade the performance and scalability of replicated multi-master systems due to multiple rounds of messages and blocking [94, 114, 76, 39]. Hence, *where* the master copies of data are placed can significantly affect system performance.

This examination of the two architectures poses the following challenging question: can I design a replicated database system that preserves the benefits of the single-master and multi-master architectural approaches while addressing their shortcomings? Such an architecture should support scalability by distributing the update load over multiple sites while also avoiding expensive multi-site transaction coordination. That is, the new architecture should (i) allow single-site transaction execution for *all transactions* (read-only and update), and (ii) support system scalability by distributing load of *both update and read-only transactions* by changing the location of master copies.

While the design of this new architecture brings significant benefits, it poses several research challenges. First, the system must support *dynamic* single-site transaction execution, that is, the system needs to select one site at which to execute a transaction *without* constraining execution to this same site for all transactions. Thus, the replicated system should be flexible so as to support one-site execution at *any site* in the system, thereby offering opportunities to distribute load among sites. Second, all master copies of data that a transaction needs to update as well as read need to be located at the execution site. Deciding where to locate master copies of data while ensuring that transactions see, and install, a consistent view of the data adds to the set of challenging problems to solve.

In this thesis, I address these challenges by designing and building a new replicated system called **DynaMast** [12] that provides all of the above desirable properties while addressing the deficiencies of prior approaches. DynaMast guarantees one-site transaction executions by *dynamically transferring data mastership* among sites in the distributed system while maintaining transactional consistency. I call this technique *remastering*, which also distributes update and read load among sites in the system.

DynaMast decides the master location of each data item using comprehensive strategies that consider data item access patterns to balance load among sites and minimize future remastering. When remastering occurs, it is efficient as DynaMast uses a lightweight protocol that exploits the presence of replicas to transfer mastership using metadata-only operations. These design decisions enable DynaMast to significantly reduce transaction latency compared to both the single-master and multi-master replicated architectures. Moreover, I show DynaMast's ability to flexibly and dynamically select physical transaction execution sites in the distributed system by learning workload data access patterns, thereby delivering superior performance in the face of changing workloads.

The contributions of this work are three-fold:

- A novel replication protocol that efficiently supports dynamic mastership transfer to guarantee single-site transactions while maintaining well-understood and established transactional semantics.
- I propose remastering strategies that learn workload data-access patterns and exploit them to remaster data adaptively. The strategies transfer the mastership of data among sites to minimize future remastering, which in turn reduces transaction processing latency.
- I empirically compare DynaMast with single-master and multi-master architectures, demonstrating DynaMast's superiority on a range of workloads.

1.3.2 Automatic Physical Design for Distributed DBMS

Recall that data replication and partitioning are two primary means of scaling out a DBMS. The chosen data replication and partitioning schemes for a distributed database form its *physical design*. Constructing a distributed physical design includes making the following key decisions (i) what the data partitions should be, (ii) which data partitions to replicate, and (iii) where master copies and their replicas should be placed. These decisions, in turn, determine the sites where transactions execute.

For example, the partitioned database is a well-known distributed physical design [96, 187] that distributes partitioned data such that transactions execute at different sites, thereby spreading both the update and read load over multiple sites in the distributed system. Similarly, replicating data to multiple sites [196, 205], which allows transactions to execute on data copies, represents another distributed physical design. Each of these designs presents performance trade-offs, that a system administrator or offline tool must select based on prior workload knowledge. However, these static decisions cannot adapt to workload changes or are not suitable if extensive information about the workload, e.g., access patterns, is not available *a priori*. Thus, static distributed physical designs fall short in delivering good performance in the presence of hotspots, changing workloads or when workload information is not available a priori to inform physical design decisions.

Chapter 4 presents **MorphoSys** [13], a distributed database system that I have designed and built that makes decisions *automatically* on how to partition data, what to replicate, and where to place these partitioned and replicated data. MorphoSys learns from workload locality characteristics what the partitions should be, and where to place master and replica partitions, *dynamically.* This dynamism frees the system from having to know workload access patterns a priori, allowing MorphoSys to *change* or metamorphosize the distributed physical design on the fly. Moreover, to avoid expensive multi-site transaction coordination, MorphoSys dynamically alters its physical design to co-locate co-accessed data and to guarantee single-site transaction execution.

The contributions of this work are four-fold:

- A system that dynamically constructs physical designs using a set of physical design change operators.
- A novel concurrency control algorithm and update propagation protocol to support efficient execution of transactions and physical design changes.
- A cost model that drives physical design decisions to improve transaction processing performance by learning and exploiting workload patterns.
- An extensive evaluation that establishes MorphoSys' efficacy to deliver superior performance over prior approaches.

1.3.3 Adaptive Storage for Hybrid Database Workloads

As discussed, DBMSs optimized for OLTP workloads utilize row-based data formats while OLAP focused systems use column-based data layouts. To show that neither row-oriented nor column-oriented storage format is optimal for processing both transactional and analytical workloads concurrently, I conducted microbenchmark experiments. The microbenchmarks update 100 rows, or perform a scan of 10,000 rows over 1 column out of 10 with 10% selectivity or 100% selectivity.

As Figure 1.4 shows, a row-oriented format supports updates at half the latency of the column-oriented storage. However, column-oriented storage can support analytical (scan) operations 7× faster than row-oriented storage. This experiment demonstrates the performance impact, and importance, of storage format on a hybrid workload. Neither a row format nor a column format alone is suited for a workload consisting of both transactional updates and analytical queries, as the performance of one type of the workload suffers due to the static format of the data.

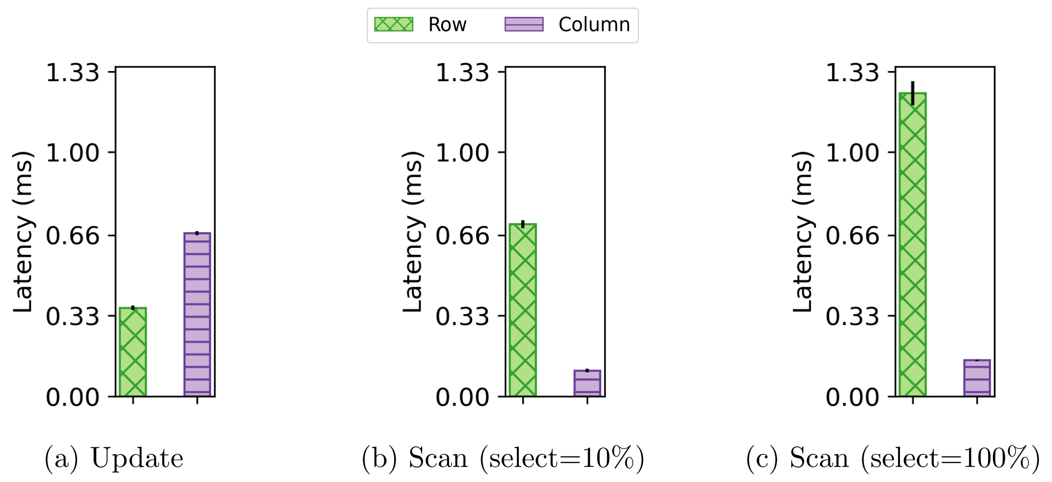


Figure 1.4: The average latency of 100 updates (1.4a) and scans of 10,000 data items (1.4b and 1.4c) on row and column formats.

To mitigate the effect of storage formats on latencies, the traditional architecture for *hybrid* transaction/analytical processing *(HTAP)* workloads periodically migrates new data from an OLTP system to an OLAP system using extract-transform-load (ETL) utilities [144]. While this procedural transformation allows organizations to continue executing OLTP and OLAP workloads concurrently, periodic data migration results in recent (OLTP) updates that are absent in the OLAP system. Thus, organizations cannot obtain real-time insights from their data [26]. Modern HTAP systems address these concerns by storing data in *both* OLTP and OLAP formats and executing queries across both formats in a single integrated system [26, 71, 84, 125, 25]. However, replicating all data within a system consumes at least double the storage resources, particularly for expensive in-memory processing, and requires costly maintenance across replicas to guarantee data freshness and consistency.

Although modern HTAP systems improve performance compared to ETL pipelines, there are four essential aspects of system design that no system has considered integrally. First, a scale-out, distributed HTAP system can meet the data storage and processing demands of large scale workloads that exceed the capabilities of a single node [84, 125, 25, 206, 68]. Additionally, the overheads of distributed transaction coordination mean that applying standard partitioning techniques to existing single-node systems limits scalability. Moreover, scale-out systems allow for parallel execution both within and among queries.

Second, HTAP systems should selectively store the same data item in different formats (e.g., row and column) and in different storage tiers (e.g., memory and disk), simultaneously if desirable, using an efficient replication scheme [158, 125, 25]. HTAP systems that mandate data storage in a single format or storage tier at a time suffer when transaction processing and analytics occur concurrently. Such concurrent workloads frequently occur as a consequence of real-time data analysis (as in the e-commerce example), fraud detection, IoT, and logistics domains [26, 144, 153, 152, 120].

Third, HTAP systems should leverage layout and storage-specific optimizations, such as per-column sort orders and compression [9, 89, 90, 213, 8, 10, 97], which reduce CPU and storage overheads, or row splitting [87] to avoid conflicts. Using these optimizations enables HTAP systems to match the performance of specialized systems.

Fourth, based on the workload, HTAP systems should adapt their physical storage layout autonomously, that is, without manual intervention [26, 151]. The rising complexity of workloads means that system administrators cannot decide on a single efficient data layout that both effectively utilizes resources and reduces transaction latency [26, 13]. Furthermore, when workloads change, a layout that works well for one workload is unlikely to work well for another type of workload, as depicted in Figure 1.4.

Chapter 5 of this thesis presents **Proteus** [14, 15], a distributed database system that delivers the aforementioned requirements of an *integral* HTAP system. Proteus *adaptively* stores data in *multiple formats* and *storage tiers* to support HTAP workloads efficiently. *Based on the workload*, Proteus makes adaptive storage decisions *autonomously* and leverages *layout-specific optimizations* that enable it to *concurrently* achieve OLTP throughput comparable to row-oriented storage systems and OLAP query latencies that are on par with column-oriented storage systems. Consequently, Proteus significantly outperforms other state-of-the-art distributed HTAP systems.

The contributions of this work are four-fold:

- The case for adaptive storage for HTAP workloads.
- The architecture and design of adaptive storage in Proteus and how transaction and query execution is supported in this environment.
- A model to learn workload patterns and make cost-based layout decisions to support high throughput transactions and low latency queries for HTAP workloads.
- An extensive experimental evaluation that demonstrates the effectiveness and performance benefits of Proteus.

Table 1.1: A summary of the contributions in this thesis.

Summary of Contributions **System**	**Context**	**What is Adapted?**	**Techniques for Efficient Execution**	**How are Decisions Made?**
DynaMast (Chapter 3) [12]	Replicated DBMS	Master location of data	Remastering protocol	Heuristic Strategies
MorphoSys (Chapter 4) [13]	Distributed DBMS	Distributed Physical Design (data replication, data partitioning, data mastership)	Physical Design Change Operators Partition-based multi-version concurrency control	Learned Cost Model
Proteus (Chapter 5) [14, 15]	HTAP DBMS	Storage layout of data (storage format, tier, replication, partitioning, mastership)	Storage Layout Changes Storage Aware Operators	Estimating access latencies and arrivals

1.3.4 Summary of Contributions

Table 1.1 summarizes the contributions of this thesis by system. Each system operates in a different context and adapts different aspects of the DBMS. Moreover, each system describes novel techniques to execute transactions and adaptations efficiently and techniques to make adaptation decisions. Collectively, this work is an exciting step towards the vision of autonomous database systems that allow users to specify only the data to store and the queries to execute, leaving the system to choose the storage and execution mechanisms [45, 151].

1.4 Thesis Outline

The rest of this thesis is organized as follows. Chapter 2 introduces an adaptive distributed database system architecture that is used in the design of the systems presented in this thesis. Chapters 3, 4, and 5 present these three systems, DynaMast, MorphoSys and Proteus, respectively. Related work is presented in Chapter 6. I present the conclusion of this thesis and directions for future research in Chapter 7.

Chapter 2

Architecture

To support both efficient transaction execution and adaptation in a distributed DBMS, I propose a shared-nothing [181] two-tier architecture consisting of an *adaptation advisor* and *data sites*, as shown in Figure 2.1. Data sites and adaptation advisors execute on separate physical nodes (servers), each equipped with local memory and secondary storage. Communication between nodes occurs over a local area network.

Clients submit transactions to the adaptation advisor that decides *where* (at which site(s)) and *how* (execution plan) a transaction will execute. The adaptation advisor does so by: (i) identifying the location of the data that the transaction needs to access, which it collects and stores as metadata, (ii) initiating changes to how and where data is stored based on the submitted transaction, (iii) forwarding the transaction to the relevant data sites for execution, and (iv) tracking statistics about submitted transactions to model the workload for use when making its decisions.

Deciding where and how to store data for every data item results in a large set of possible decisions to make and manage. Thus, data items are grouped together into data *partitions* and the decision of how and where to store data is made on a per-partition basis. A partition is a contiguous range of one or more rows and columns based on the primary key of each row (*row_id*) and column identifiers. A partition may range from a single cell to an entire table, however every cell belongs to exactly one partition. The master, or updateable copy, of a partition is located at a single data site in the system, though different partitions may be mastered at different sites. Partitions may have read-only replicas at other data sites. Updates to the partition must always occur at the site that masters the partition, while reads to the partition can occur at either master or replica sites. It is the responsibility of the adaptation advisor to track this partition metadata.

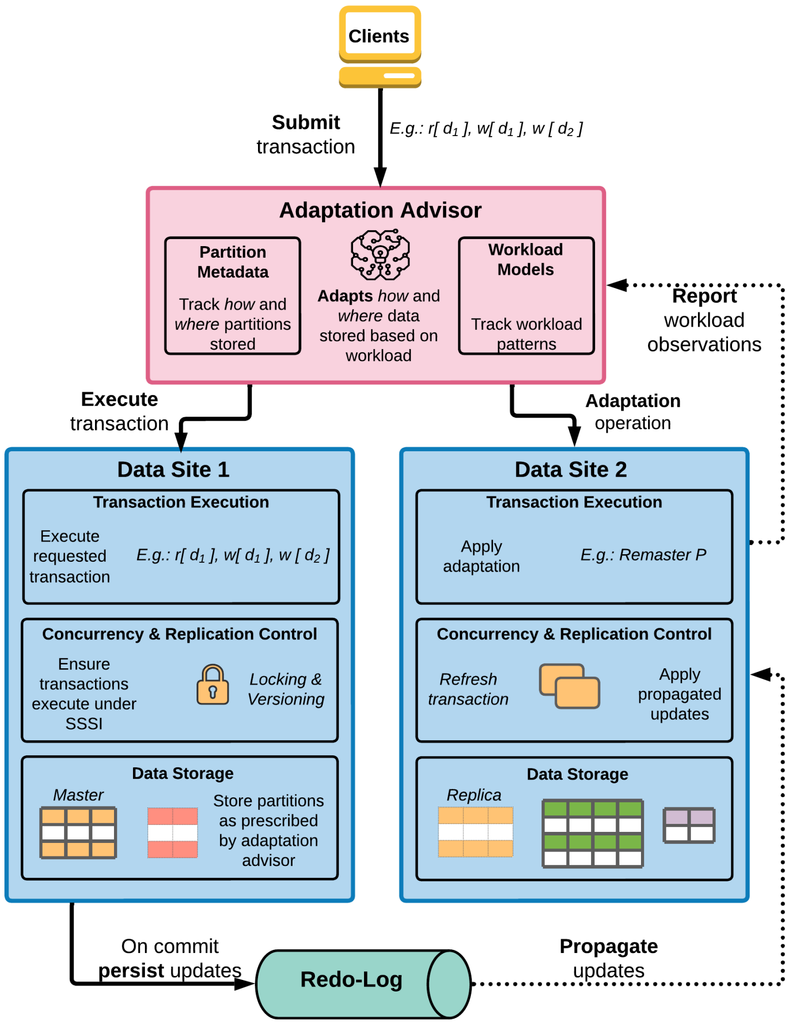

Clients
Submit transaction
E.g.: r[d1], w[d1], w [d2]
Adaptation Advisor
Partition Metadata
Track how and where partitions stored
Adapts how and where data stored based on workload
Workload Models
Track workload patterns
Report workload observations
Execute transaction
Adaptation operation
Data Site 1
Transaction Execution
Execute requested transaction
E.g.: r[d1], w[d1], w [d2]
Concurrency & Replication Control
Ensure transactions execute under SSSI
Locking & Versioning
Data Storage
Master
Store partitions as prescribed by adaptation advisor
Data Site 2
Transaction Execution
Apply adaptation
E.g.: Remaster P
Concurrency & Replication Control
Refresh transaction
Apply propagated updates
Data Storage
Replica
On commit persist updates
Redo-Log
Propagate updates

Figure 2.1: An architecture for an adaptive distributed database management system. The adaptation advisor receives requests from clients, and routes them to data sites for execution. To improve system performance, the adaptation advisor adapts how and where the system stores data.

Data sites store data as prescribed by the adaptation advisor and execute transactions. Data sites coordinate transactions amongst each other to ensure that transactions observe a consistent state. Data sites persist transactions by committing their updates to a redo-log (Chapter 2.2). If a data site contains a replica of data mastered at another site, the replica subscribes to updates to the data via the log, and lazily (asynchronously) applies the update as a refresh transaction consistently.

2.1 System Model

The system architecture supports *transactions* and provides strong-session snapshot isolation (SSSI) [56], which strengthens the guarantees provided by *snapshot isolation* (SI). SI is a standard [31, 52, 16] and popular isolation level in both single node DBMSs [189, 160] and distributed DBMSs [84, 206, 179, 32], while SSSI is common for distributed and replicated database systems [47, 103, 35, 107]. SSSI strengthens SI by preventing transaction inversion and ensures that client transactions always see all updates from their previous transactions. Under SSSI, every transaction T is assigned a begin timestamp such that T sees the updates made by transactions with earlier commit timestamps. SSSI systems select begin timestamps that are larger than the latest commit timestamps of prior transactions submitted by the client to prevent transaction inversion. SSSI systems also ensure that if two transactions update the same data item and have overlapping timestamps, only one transaction successfully commits [31]. Additionally, for every update transaction T, its corresponding refresh transaction $R(T)$ is applied to replicas in an order that ensures transactions observe a consistent snapshot of data.

Both the data site's concurrency control and adaptation advisor make use of transactional read/write information. For analytical queries, clients determine this information from the columns accessed in each table necessary to execute each query. For OLTP transactions, clients have this information based on primary keys, or if needed, execute reconnaissance queries [119, 194].

2.2 The Redo-Log

The redo-log is implemented using Apache Kafka [102] and serves as the single persistent source of truth of the database state. The redo-log also acts as an asynchronous event log between data sites. The redo-log has the following properties:

Property 1. *Log entries from a data site are delivered at least once to other data sites.*

Property 2. *Log entries written to the redo-log are persistent.*

The implementation creates creates distinct Apache Kafka [102] logs for updates from each site, which provides Properties 1 and 2.

Each data site writes all data item updates and adaptive changes to a site-specific redo-log on commit. Each entry in this log has a sequence number, which depends only on site-specific state, that can be used to de-duplicate and re-order log entries. The simplest sequence number is a site local counter that is atomically incremented on each transaction commit or adaptation.

When consuming log entries, the data site maintains state on a per-site basis. This state is (i) the next expected log entry sequence number n, and (ii) a (priority) queue of buffered log entries, q, based on sequence number. Data sites skip log entries with sequence number k if $k < n$ or there is a log entry in q with sequence number equal to k, which eliminates duplicate log entries. The replication layer consumes log entries from q if there is an entry with sequence number equal to n, the next expected entry. If such an entry exists, it must be the minimum, an efficient operation for a priority queue. When the replication layer consumes this entry, it increments n. Hence, from the perspective of the replication layer, per-site log entries are consumed exactly once and are ordered by log entry sequence number.

2.3 Fault Tolerance and Recovery

The distributed database system uses redo-logs as the basis of recovery and to guarantee fault-tolerance.

If a data site fails, it first recovers to an initial snapshot state by consulting an existing replica or stored checkpoint of data and copying this state. Next, the data site begins replaying the persistent redo-logs (Property 2) from the log entry sequence number associated with this initial snapshot state. Once the recovered data site applies all updates in its redo-log, then it has recovered and can begin serving transactions. The data site will continue to apply propagated updates from other data sites, becoming indistinguishable from a data site lagging in applying propagated updates. Recovery at a data site is guaranteed to complete because the site's redo-log is bounded in size and cannot grow during the recovery process. A data site cannot execute any new transactions while failed, or when recovering, hence it cannot add to its redo-log.

The adaptation advisor stores metadata about how and where data is stored at the data sites, such as whether a stored partition is a master or replica copy. Thus, to recover a failed adaptation advisor, the advisor must discover this metadata. The advisor contacts each data site to determine what data is stored there. The adaptation advisor cannot make any changes while failed, or when recovering. The adaptation advisor can only reload the location of data items and whether they are replicas or a master copy.

If either: (i) at least one copy of each data partition is available or (ii) a checkpoint is accessible, then the system can recover from multiple simultaneous failures. Each failed data site can recover independently of others, and once all data sites have recovered, the adaptation advisor can subsequently recover.

Finally, the prior discussion assumes that a data site can be recovered. If the site cannot be recovered, then any of its mastered data items must be remastered to other data sites. To do so, a lease mechanism can be used [69]. Specifically, the data site can periodically signal its continued ability to store and manage data as part of periodically reporting workload observations to the adaptation advisor (Figure 2.1). If a data site fails to renew its lease, then the adaptation advisor revokes the lease and treats the data site as failed, which stops routing transactions to the site. For each data item to be remastered, the adaptation advisor must select the new master data site and follow a modified recovery procedure. Specifically, the new site should redo all updates to the data item stored in the failed site's persistent redo-log (Property 2). Once all data item updates are redone, the site can become the master of the data item.[1]

2.4 Summary

In this Chapter, I presented a two-tiered architecture for an adaptive distributed database system. The two-tiered architecture consists of an adaptation advisor, which decides how and where to store data, and data sites, which store data and execute transactions. The architecture is also well-suited to support the strong-session snapshot isolation level, a popular and intuitive isolation level for distributed database systems. Finally, the architecture is fault-tolerant by using redo-logs as the mechanism for both replication and recovery. In the subsequent Chapters, I use this architecture as a basis for all three systems described in this thesis.

[1]This protocol is equivalent to performing only the `grant` operation described in Chapter 3.

Chapter 3

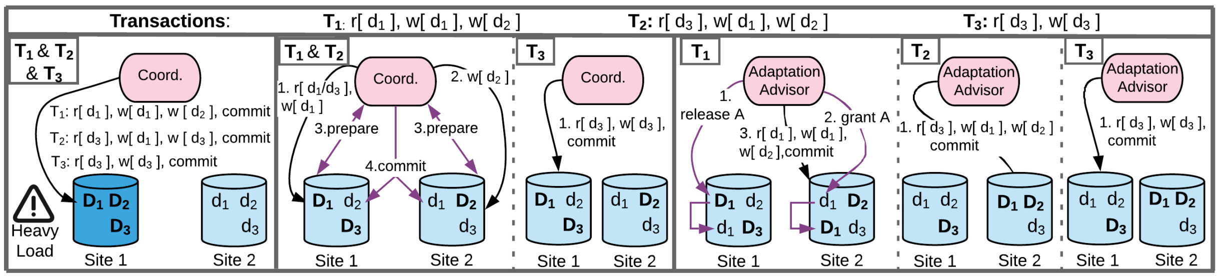
DynaMast: Adaptive Dynamic Mastering for Replicated Systems

Recall from Chapter 1 that the single-master architecture guarantees that transactions execute at a single site thereby avoiding expensive distributed coordination. In contrast, the multi-master or partitioned architecture, distributes master copies of data items to multiple sites [184]. However, the distribution of master copies results in transactions updating data at multiple sites [53, 76, 94], requiring expensive distributed commit protocols, such as two-phase commit, to ensure transactional consistency and atomicity. Unfortunately, such commit protocols significantly degrade the performance and scalability of replicated multi-master systems due to multiple rounds of messages and blocking [94, 114, 76, 39]. Hence, *where* the master copies of data are placed can significantly affect system performance.

In this Chapter, I present **DynaMast**, a replicated DBMS that adaptively transfers the mastership of data items among sites to guarantee single-site transactions. This dynamic transfer of mastership provides the advantages of both the multi-master architecture — load distribution – and the single-master architecture — single site transactions.

3.1 Background

I now discuss the limitations of the single-master and multi-master replication approaches and illustrates the benefits of dynamic mastering.


Transactions:
T1: r[d1], w[d1], w[d2]
T2: r[d3], w[d1], w[d2]
T3: r[d3], w[d3]
T1 & T2 & T3
Coord.
T1: r[d1], w[d1], w [d2], commit
T2: r[d3], w[d1], w [d3], commit
T3: r[d3], w[d3], commit
Heavy Load
Site 1
Site 2
T1 & T2
1. r[d1/d3], w[d1]
2. w[d2]
3.prepare
4.commit
T3
1. r[d3], w[d3], commit
T1
Adaptation Advisor
1. release A
2. grant A
3. r[d1], w[d1], w[d2],commit
T2
1. r[d3], w[d1], w[d2] commit

(a) Single-Master (b) Multi-Master (c) Dynamic Mastering

Figure 3.1: An example illustrating the benefits of dynamic mastering over single-master and multi-master replicated architectures. Uppercase bolded letters represent master copies of data items, with read-only replicas distributed to the other sites. Single-master bottlenecks the master site as all transactions execute there. Multi-master requires a distributed commit protocol for both transactions T_1 and T_2 as the write set is distributed. In dynamic mastering, d_1 is remastered before T_1 executes, allowing T_1 and T_2 to execute as single-site transactions at Site 2. The load is distributed by executing T_3 at Site 1.

3.1.1 Limitations of Single-Master & Multi-Master Architectures

Replicated single-master systems route all update transactions to a single site, eliminating multi-site transactions but overloading that site. Figure 3.1a illustrates this problem by example; a client submits three update transactions that all execute on the master copies (indicated by bold, uppercase letters) at the single-master site (Site 1). Hence, the update workload overloads the single-master site as it cannot offload any update transaction execution to a replicated site (Site 2).

By contrast, the replicated multi-master architecture distributes updates among the sites to balance the load. In the example in Figure 3.1b, updates to data item d_1 and d_3 execute at Site 1, while updates to d_2 execute at Site 2. However, transactions that update both d_1 and d_2, such as T_1 are forced to execute at multiple sites (Site 1 and Site 2), requiring a distributed commit protocol to ensure atomicity. As discussed in Chapter 1, such protocols are expensive as they incur overhead from blocking while waiting for a global decision and suffer from latencies due to multiple rounds of communication. I illustrate this using the popular two-phase commit (2PC) protocol [177] in Figure 3.1b (Steps 2-4). Observe that all transactions with distributed write sets, such as T_2 in Figure 3.1b, must execute as expensive distributed transactions. Only transactions with single-site write sets, such as T_3, are free to execute as local transactions in the multi-master replicated architecture.

3.1.2 Dynamic Mastering

Figure 3.1c shows how transactions T_1, T_2 and T_3 execute using the dynamic mastering protocol, which, as in the single-master case, guarantees single-site execution while allowing the distribution of load as in the multi-master architecture. Like a multi-master architecture, observe that both sites contain master copies of data. Executing T_1 at one site requires changing the mastership of either data item d_1 or d_2. Our adaptation advisor recognizes this requirement and, without loss of generality, decides to execute the transaction at Site 2, and therefore dynamically remasters d_1 from Site 1 to Site 2. To do so, the adaptation advisor sends a `release` message for d_1 to Site 1, which releases d_1's mastership after any pending operations on d_1 complete (Step 1). Next, the adaptation advisor informs Site 2 that it is now the master of d_1 by issuing the site a `grant` message for d_1 (Step 2). In Step 3, T_1 executes unhindered at Site 2 by applying the operations and committing locally, requiring no distributed coordination. Through careful algorithmic and system design that enables remastering outside the boundaries of transactions and using metadata-only operations, DynaMast ensures that dynamic mastering is efficient (Chapter 3.2).

Remastering is necessary only if a site does not master all of the data items that a transaction will update. For example, in Figure 3.1c, a subsequent transaction T_2 also updates d_1 and d_2, and therefore executes without remastering, *amortizing* the first transaction's remastering costs. Unlike the single-master architecture (Figure 3.1a), dynamic mastering allows T_3 to execute at a different site, thereby *distributing the write load* through multiple one-site executions. Thus, it is important that the adaptation advisor *adaptively* decides where to remaster data to balance load and minimize future remastering — objectives the remastering strategy takes into account (Chapter 3.3).

I consider a model in which data is fully replicated so that every site has a copy of every data item, allowing flexibility in mastership placement.

3.2 Dynamic Mastering Protocol

Having presented an overview of remastering and its benefits, I now detail the dynamic mastering protocol and its implementation.

3.2.1 Maintaining Consistent Replicas

The dynamic mastering protocol exploits lazily maintained replicas to transfer ownership of data items efficiently. Lazily-replicated systems execute update transactions on the master copies of data, and apply updates asynchronously to replicas as *refresh transactions*.

I now describe how the dynamic mastering protocol guarantees snapshot isolation (SI) that is popularly used by replicated database systems [179, 32, 56], and later outline the provision of session guarantees on top of SI. Recall from Chapter 2.1 that supporting SSSI requires that (i) if two transactions update the same data item and have overlapping timestamps, only one transaction successfully commits, and (ii) for every update transaction T, its corresponding refresh transaction $R(T)$ is applied to replicas in an order that ensures transactions observe a consistent snapshot of data.

To apply refresh transactions in a consistent order, DynaMast track each site's state using *version vectors*. In a dynamic mastering system with m sites, each site maintains an m-dimensional vector of integers known as a *site version vector*, denoted $svv_i[\,]$ for the i^{th} site (site S_i), where $1 \leq i \leq m$. The j-th index of site S_i's version vector, $svv_i[j]$, indicates the number of refresh transactions that S_i has applied for transactions originating at site S_j. Therefore, whenever site S_i applies the updates of a refresh transaction originating at site S_j, S_i increments $svv_i[j]$. Similarly, $svv_i[i]$ denotes the number of locally committed update transactions at site S_i.

DynaMast assigns update transactions an m-dimensional transaction version vector, $tvv[\,]$ that acts as a commit timestamp and ensures that updates are applied in an order consistent with this commit timestamp across sites. When an update transaction T begins executing at site S_i, it records S_i's site version vector $svv_i[\,]$ as T's begin timestamp ($tvv_{B(T)}[\,]$). During commit, T copies $tvv_{B(T)}[\,]$ to tvv_T, increments $svv_i[i]$ and copies that value to $tvv_T[i]$. Thus, $tvv_T[\,]$ (the commit timestamp) reflects T's position in S_i's sequence of committed update transactions, while $tvv_{B(T)}[\,]$ (the begin timestamp) represents the updates visible to T when it executed.

Recall from Chapter 2.1 that SSSI prevents transaction inversion by adding session freshness rules. To enforce the client session-freshness guarantee, the system tracks each client's state using a client version vector $cvv_c[\,]$ and ensures that a client's transaction executes on data that is at least as fresh as the state last seen by the client. Specifically, transactions respect the following *freshness rules*: when a client c with an m-dimensional client session version vector $cvv_c[\,]$ accesses data from a site S_i with site version vector $svv_i[\,]$, c can execute when $svv_i[k] \geq cvv_c[k], \forall k \in (1, \ldots, m)$. After the client accesses the site, it updates its version vector to $svv_i[\,]$.

The transaction version vector and site version vector indicate when a site can apply a refresh transaction. Given a transaction T that commits at site S_i, a replica site S_j applies T's refresh transaction $R(T)$ only after S_j commits all transactions whose committed updates were read or updated by T. Formally, T *depends* on T' if T reads or writes a data item updated by T'. That is, $R(T)$ cannot execute and commit at S_j until S_j commits all transactions that T depends on. The transaction version vector ($tvv_T[\,]$) encapsulates the transactions that T depends on, while the site version vector ($svv_j[\,]$) indicates the updates committed at S_j. Hence, site S_j blocks the application of $R(T)$ until the following *update application rule* holds (Equation 3.1):

$$\Big(\bigwedge_{k\neq i} svv_j[k] \geq tvv_T[k]\Big) \wedge \big(svv_j[i] = (tvv_T[i] - 1)\big) \tag{3.1}$$

As an example of the update application rule, consider the three-sited system in Figure 3.2. In the first two steps, transaction T_1 updates a data item and commits locally at site S_1, which increments the site version vector from $[0, 0, 0]$ to $[1, 0, 0]$. Next, T_1's refresh transactions, $R(T_1)$, begin applying the updates to sites S_2 and S_3. $R(T_1)$ commits at S_3, and sets $svv_3[\,]$ to $[1, 0, 0]$ (Step 4), but site S_2 has not yet committed $R(T_1)$ (Step 5). In Step 6, transaction T_2 begins after $R(T_1)$ and commits at site S_3, and therefore sets $svv_3[\,]$ to $[1, 0, 1]$, capturing that T_2 depends on T_1. The update application rule blocks S_2 from applying $R(T_2)$ until $R(T_1)$ commits (Step 7). Without blocking the application of T_2, it would be possible for T_2's updates to be visible at site S_2 before T_1's updates have been applied, despite the fact that T_2 depends on T_1. Finally, after site S_2 applies and commits $R(T_1)$, it applies $R(T_2)$, which results in $svv_2[\,]$ being set to $[1, 0, 1]$ and ensures a globally consistent order of update application.

3.2.2 Transaction Execution and Remastering

The dynamic mastering protocol is implemented within the architecture shown in Figure 2.1 consisting of an adaptation advisor and data sites composed of a transaction execution layer, concurrency control and replication layer and data storage layer. Clients submit transactions to the adaptation advisor, which remasters data items if necessary and routes transactions to an appropriate data site. The concurrency control layer at each data site interacts with the data storage layer to maintain version vectors, apply propagated updates as refresh transactions, and handle remastering requests. The transaction execution layer processes transactions and sends transactional updates to the replication layer, which forwards them to the other sites for application as refresh transactions.

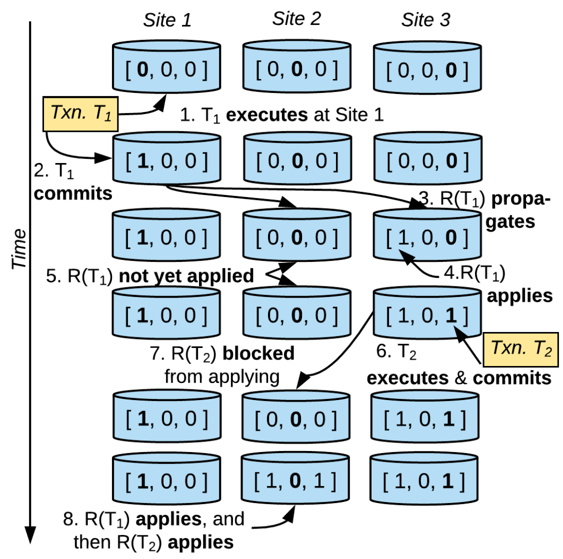


Figure 3.2: An example showing how commits and update propagation affect site version vectors.

Clients and the system components interact with each other using remote procedure calls (RPCs). Clients issue transactions by sending a `begin_transaction` request containing the transaction's write set to the adaptation advisor, which decides on the execution site for the transaction using the routing and remastering strategies discussed in Chapter 3.3. If necessary, the adaptation advisor transfers the mastership of relevant data items to the execution site via remastering.

To perform remastering (Algorithm 1), the adaptation advisor issues parallel `release` RPCs to each of the data sites that hold master copies of data items to be remastered (Line 7). When a data site receives a `release` message, it waits for any ongoing transactions writing the data to finish before releasing mastership of the items and responding to the adaptation advisor with the data site's version vector. Immediately after a `release` request completes, the adaptation advisor issues a `grant` RPC to the site that will execute the transaction (Line 8). This data site waits until updates to the data item from the releasing site have been applied to the point of the release. The data site then takes mastership of the granted items and returns a version vector indicating the site's version at the time it took ownership. After all necessary `grant` requests complete, the adaptation advisor computes the element-wise max of the version vectors that indicates the minimum version that the transaction must execute on the destination site (Line 9). Finally, the adaptation advisor notifies the client of the site that will execute its transaction and this minimum

Algorithm 1 Remastering Protocol

```
Require: Transaction $T$'s write set $w$
Ensure: The site $S$ that will execute the update transaction $T$, and version vector $out\_vv$
    indicating $T$'s begin version
 1: $S$ = determine_best_site($w$) // execution site
 2: $data\_item\_locs$ = group_by_master_loc($w$)
 3: $out\_vv = \{0\} \times m$ //zero_vector
 4: // In parallel:
 5: for ($site\ S_d$, $data\_item\ d$) $\in data\_item\_locs$ do
 6:    if $S_d \neq S$ then
 7:       // $S_d$ currently masters $d$
 8:       $rel\_vv$ = send_release($S_d, d$)
 9:       $grant\_vv$ = send_grant($S$, $d$, $rel\_vv$)
10:       $out\_vv$ = elementwise_max($out\_vv, grant\_vv$)
11:    end if
12: end for
13: return ($out\_vv$, $S$)
```

version vector. `release` and `grant` are executed as transactions, and consequently a data item waits to be remastered while being updated.

Appendix A.1 formalizes the correctness of DynaMast's remastering protocol and how it preserves SSSI through detailed proofs.

Remastering ensures correctness in the presence of failures, with the key event being when the failure happens with respect to writes to the redo-log. If a site fails before the `release` writes to the log, or after the `grant` writes to the log, then recovery proceeds as described in Chapter 2.3. If failure at the new master site happens between these log writes, then no site will master these data items upon recovery. The adaptation advisor may then select a new master site and send a `grant` request. Thus, at all times, at most one site masters a data item at a time.

Parallel execution of `release` and `grant` operations greatly speed up remastering, reducing waiting time for clients. Clients begin executing as soon as their write set is remastered, benefiting from the remastering initiated by clients with common write sets. A client then submits transactions directly to the data site. Client `commit/abort` operations are submitted to the data site. Note that since data is fully replicated, clients need not synchronize with each other unless they are waiting for a `release` or `grant` request. Further, read-only transactions may execute at any site in the system without synchronization across sites.

A key performance *advantage* is that coordination through remastering takes place outside the boundaries of a transaction, and therefore does not block running transactions. Once a transaction begins at a site, it executes without any coordination with any other sites, allowing it to commit or abort unilaterally. By contrast, in a multi-master architecture, changes made by a multi-site transaction are not visible to other transactions because the database is uncertain about the global outcome of the distributed transaction. Thus, distributed transactions block local transactions further increasing transaction latency [94]. To illustrate this benefit, consider a transaction T_4 that updates B concurrently with T_1 in our example from Figure 3.1. T_4's update to B can occur while A is being remastered, unlike during 2PC in multi-master that blocks T_4 while the global outcome of T_1 is unknown.

Thus, remastering operations (i) change only mastership location metadata, (ii) occur outside the boundaries of transactions, (iii) do not physically copy master data items, and (iv) allow one-site transaction execution once a write set is localized. These features make our dynamic mastering protocol *lightweight* and *efficient*, resulting in significant performance advantages over existing approaches (Chapter 3.5).

3.3 Adaptation Advisor Strategies

In Chapter 3.2, I described the dynamic mastering architecture and the importance of adaptive remastering decisions. The DynaMast system implements this dynamic mastering architecture and efficiently supports it using comprehensive transaction routing and remastering strategies. I will now describe these strategies in detail, deferring descriptions of their implementation to Chapter 3.4.

Transaction routing and remastering decisions play a key role in system performance. Routing and remastering strategies that do not consider load balance, data freshness, and access patterns for data items can place excessive load on sites, increase latency waiting for updates, and cause a ping-pong effect by repeatedly remastering the same data between nodes. Thus, DynaMast uses comprehensive strategies and *adaptive* models to make transaction routing and remastering decisions.

3.3.1 Write Routing and Remastering

When the adaptation advisor receives a write transaction request from client c, it first determines whether the request requires remastering. If all items the transaction wishes to update are currently mastered at one of the sites, then the adaptation advisor routes the transaction there for local execution. However, if these items' master copies are distributed across multiple sites, the adaptation advisor co-locates the items via remastering before transaction execution. DynaMast makes remastering decisions prudently: data is remastered only when necessary, employing strategies that choose a destination site that minimizes the amount of remastering in the future.

These remastering strategies consider load balance, site freshness, and data access patterns. DynaMast uses a linear model that captures and quantifies these factors as input features and outputs a score that represents an estimate of the expected benefits of remastering to a site. Concretely, DynaMast computes a score for each site, indicating the benefits of remastering the transaction's write set there and then remasters these data items to the site that obtained the highest score.

Balancing Load

Workloads frequently contain access skew, which if left unaddressed can result in resource under-utilization and performance bottlenecks [191]. Consequently, DynaMast's remastering strategy balances data item mastership allocation among the sites according to the write frequency of the items, which in turn balances the load.

When evaluating a candidate site S as a destination for remastering, DynaMast considers both the *current* write load balance and the *projected* load balance if it were to remaster to S the items that the transaction wishes to update. For a system mastership allocation X, I express the write balance as the *distance from perfect write load balancing* (where every one of the m sites processes the same volume of write requests):

$$f_{balance\_dist}(X) = \sqrt{\sum_{i=1}^{m} \left(\frac{1}{m} - freq(X_i) \right)^2}$$

where $freq(X_i) \in [0,1]$ indicates the fraction of write requests that would be routed to site i under mastership allocation X. If each site receives the same fraction of writes ($\frac{1}{m}$), then $f_{balance\_dist}(X) = 0$; larger values indicate larger imbalances in write load.

DynaMast uses $f_{balance\_dist}$ to consider the change in load balance if it were to remaster the items a transaction T wishes to update to a candidate site S. Let B be the current mastership allocation and $A(S)$ be the allocation resulting from remastering T's write set to S. The change in write load balance is computed as:

$$f_{\Delta balance}(S) = f_{balance\_dist}(B) - f_{balance\_dist}(A(S)) \tag{3.2}$$

A positive value for $f_{\Delta balance}(S)$ indicates that the load would be more balanced after remastering to site S; a negative value indicates that the load would be less balanced.

Although $f_{\Delta balance}(S)$ gives an indication of a remastering operation's improvement (or worsening) in terms of write balance, it does not consider how balanced the system *currently is*. If the current system is balanced, then unbalancing it slightly, in exchange for less future remastering, may yield better performance. However, for a system that is already quite imbalanced, re-balancing it is important. DynaMast incorporates this information into a scaling factor $f_{balance\_rate}(S)$ that reinforces the importance of balance in routing decisions:

$$f_{balance\_rate}(S) = \max\left(f_{balance\_dist}(B), f_{balance\_dist}(A(S))\right) \tag{3.3}$$

DynaMast combines the change in write load balance, $f_{\Delta balance}$, with the balance rate, $f_{balance\_rate}$, to yield an overall balance factor. This factor considers both the magnitude of change in write load balance and the importance of correcting it:

$$f_{balance}(S) = f_{\Delta balance}(S) \cdot \exp\left(f_{balance\_rate}(S)\right) \tag{3.4}$$

Estimating Remastering Time

After a candidate site S is chosen as the remastering destination for a transaction, the `grant` request blocks until S applies the refresh transactions for all of the remastered items. Additionally, the transaction may block at S to satisfy session-freshness requirements. Thus, if S lags in applying updates, the time to remaster data and therefore execute the transaction increases.

DynaMast's strategies estimate the number of updates that a candidate site S needs to apply before a transaction can execute by computing the dimension-wise maximum of version vectors for each site S_i from which data will be remastered and client c's version vector ($cvv_c[\,]$). DynaMast subtracts this vector from the current version vector of S and perform a dimension-wise sum to count the number of necessary updates, expressed as:

$$f_{refresh\_delay}(S) = \left\|\max\left(cvv_c[\,], \max_i\left(svv_i[\,]\right)\right) - svv_S[\,]\right\|_1 \tag{3.5}$$

Co-locating Correlated Data

Data items are often correlated; a particular item may be frequently accessed with other items according to relationships in the data [37, 172, 53]. Thus, DynaMast considers the effects of remastering data on current and subsequent transactions. DynaMast's strategies remaster data items that are frequently written together to one site, which optimizes for current and subsequent transactions with one remastering operation. The goal of co-accessed data items sharing the same master site is similar to the data partitioning problem [172, 53], solutions to which typically model data access as a graph and perform computationally expensive graph partitioning to decide on master placement. Instead, the adaptation advisor's strategy uses a heuristic that promotes mastership co-location based on data access correlations.

DynaMast considers two types of data access correlations: data items frequently written together within a transaction (intra-transaction access correlations) and items indicative of future transactions' write sets (inter-transaction access correlations). In the former case, DynaMast wishes to keep data items frequently accessed together mastered at a single site to avoid remastering for subsequent transactions (ping-pong effect). In the latter case, DynaMast anticipates future transactions' write sets and preemptively remaster these items to a single site. Doing so avoids waiting on refresh transactions to meet session requirements when a client sends transactions to different sites. Considering both of these cases enables DynaMast to rapidly co-locate master copies of items that clients commonly access together.

To decide on master co-location, DynaMast exploits information about intra-transactional data item accesses. For a given data item d_1, DynaMast tracks the probability that a client will access d_1 with another data item d_2 in all transactions. DynaMast tracks this probability as a conditional probability $P(d_2|d_1)$. That is, given that a transaction accesses d_1, what is the likelihood that d_2 is also accessed in that transaction. For a transaction with write set w that necessitates remastering and a candidate remastering site S, DynaMast considers all d_1 in the write set w, all data items d_2, and computes the *intra-transaction localization factor* as:

$$f_{intra\_txn}(S) = \sum_{d_1 \in w} \sum_{d_2} P(d_2|d_1) \times \texttt{single\_sited}(S, \{d_1, d_2\}) \tag{3.6}$$

where `single_sited` returns 1 if remastering the write set to S would place the master copies of both data items at the same site, -1 if it would allocate the master copies of d_1 and d_2 to different sites, and 0 otherwise (no change in co-location). Thus, `single_sited` encourages remastering data items to sites such that future transactions would not require remastering. Importantly, Equation 3.6 considers the effect of remastering on data items that may not be in the transaction (d_2). DynaMast normalizes the benefit of remastering these items by the likelihood of data item co-access. Thus, DynaMast need only consider data items d_2 for a given d_1 such that $P(d_2|d_1) > 0$, and therefore considers only d_2 that were in a transaction with d_1, not all possible d_2.

Consequently, a positive f_{intra_txn} score for site S indicates that remastering the transaction's write set to S will improve data item placements overall and reduce future remastering.

DynaMast also tracks inter-transactional access correlations, which occur when a client submits a transaction that accesses item d_2 within a short time interval of accessing a data item d_1. This interval, Δt, is configured based on inter-transactional arrival times and denote the probability of this inter-transactional access as $P(d_2|d_1; T \leq \Delta t)$. For a transaction with write set w and candidate remastering site S, DynaMast computes the *inter-transaction localization factor* similarly to Equation 3.7, but normalize with inter-transactional likelihood:

$$f_{inter\_txn}(S) = \sum_{d_1 \in w} \sum_{d_2} P(d_2|d_1; T \leq \Delta t) \times \texttt{single\_sited}(S, \{d_1, d_2\}) \tag{3.7}$$

$f_{inter\_txn}(S)$ quantifies the effects of remastering the current transaction's write set to candidate site S with respect to accesses to data items in the future.

Putting It All Together

Each of the previously described factors come together to form a comprehensive model that determines the benefits of remastering at a candidate site. When combined, features complement each other and enable the adaptation advisor to find good master copy placements.

$$\begin{aligned} f_{benefit}(s) =& w_{balance} \cdot f_{balance}(s) + w_{delay} \cdot f_{refresh\_delay}(s) + \\ & w_{intra\_txn} \cdot f_{intra\_txn}(s) + w_{inter\_txn} \cdot f_{inter\_txn}(s) \end{aligned} \tag{3.8}$$

DynaMast combines the scores for site S in Equations 3.4 through 3.7 using a weighted linear model (Equation 3.8), and remaster data to the site that obtains the highest score.

3.3.2 Read Routing

Site load and timely update propagation affect read-only transaction performance as clients wait for sites to apply updates present in their session. Thus, DynaMast routes read-only transactions to sites that satisfy the client's session-based freshness guarantee. DynaMast randomly chooses a site that satisfies this guarantee, which both minimizes blocking and spreads load among sites.

3.4 The DynaMast System

DynaMast implements the dynamic mastering architecture and consists of an adaptation advisor and data sites. The adaptation advisor uses the remastering strategies from Chapter 3.3. System components communicate via the Apache Thrift RPC library [178].

3.4.1 Data Sites

A data site is responsible for executing client transactions. Integrating the transaction execution, data storage and concurrency control layers improves system performance by avoiding concurrency control redundancy within the data site while minimizing logging and replication overheads. The replication layer propagates updates among data sites through writes to a durable *log* that also serves as a persistent redo-log (Chapter 2.3). For fault tolerance and to scale update propagation independently of the data sites, DynaMast uses Apache Kafka [102] to store logs and transmit updates.

Data Storage and Concurrency Control

The data site stores records belonging to each relation in a row-oriented in-memory table using the primary key of each record as an index. DynaMast system supports reading from snapshots of data using multi-version concurrency control (MVCC), similar to Microsoft's Hekaton engine [105, 57], to exploit SSSI [56]. The data site stores multiple versions (by default four, as determined empirically) of every record, which are called *versioned records*, that are created when a transaction updates a record. Transactions read the versioned record that corresponds to a specific snapshot so that concurrent writes do not block reads [57]. To avoid transactional aborts on write-write conflicts, DynaMast uses locks to mutually exclude writes to records, which is simple and lightweight.

Update Propagation

Recall from the update application rule (Chapter 3.2.1) that when an update transaction T commits at site S_i, the site version vector index $svv_i[i]$ is atomically incremented to determine commit order and the transaction is assigned a commit timestamp (transaction version vector) $tvv_T[\ ]$. Site S_i's replication layer serializes $tvv_T[\ ]$ and T's updates and writes them to S_i's log. Each data site subscribes to updates from logs at other sites. When another site S_j receives T's propagated update(s) from S_i's log, S_j's replication layer deserializes the update, follows the update application rules from Chapter 3.2.1, and applies T's updates as a refresh transaction by creating new versioned records. Finally, the replication layer makes the updates visible by setting $svv_j[i]$ to $tvv_T[i]$.

Avoiding the Long Fork Anomaly

DynaMast enforces a total commit order that transactions and data sites follow to avoid the long fork anomaly [179]. The long fork anomaly is allowed under *parallel* snapshot isolation, but not under snapshot isolation [31]. Hence, long forks cannot occur in strong session snapshot isolation. The long fork anomaly occurs as a consequence of the write skew anomaly, which is allowed under all forms of snapshot isolation. If transactions T_1 and T_2 experience write skew, then the long fork anomaly occurs if subsequent transactions can possibly observe both states: T_1's effects but not T_2's effects, and T_2's effects but not T_1's effect. If the long fork anomaly is eliminated, the system can only observe one of: T_1's effects but not T_2's effects, or T_2's effects but not T_1's effects. That is, there is a total commit order: either T_1 commits before T_2 or T_2 commits before T_1.

DynaMast uses the adaptation advisor to enforce the total commit order. The adaptation advisor tracks sites version numbers. When transactions are routed to data sites, these versions numbers are used to enforce a total commit order as part of the begin and commit timestamp information. Hence when two transactions T_1 and T_2 give rise to write skew, as allowed in SI, where say T_2 commits after T_1, T_2 will have a greater commit timestamp than T_1. Hence, subsequent transactions observe one of three database states: (i) the initial state in which no transactions have committed, (ii) T_1's update (but not T_2's), or (iii) both T_1 and T_2's updates. Transactions cannot see a state that includes T_2's update but not T_1's update; thus, long fork is avoided.

DynaMast's concurrency control algorithm is presented in Algorithm 2. DynaMast initializes a transaction's begin and commit timestamp to the client version vector to ensure the transaction observes state at least as up to date as the latest previously observed state from the same session (Line 3). To ensure transactions respect a total commit order, the

Algorithm 2 DynaMast's Concurrency Control Protocol

```
Require: Transaction $T$'s sorted write set $\{p_w\}$, session
    timestamp $cvv_c[\,]$, and a site $S_i$ where $T$ will execute
Ensure: The updated session timestamp
 1: // Begin
 2: // At the adaptation advisor
 3: $tvv_{B(T)}[\,] = cvv_c[\,]$
 4: for each $S_j \in m$ sites do
 5:     read lock $S_j$ metadata
 6:     $tvv_{B(T)}[j] = svv_j[j]$
 7: end for
 8: unlock $\{S_j\}$ metadata
 9: // At data site $S_i$
10: // wait for $S_i$'s version $svv_i[\,]$ to reach $tvv_{B(T)}[\,]$
11: for $p \in \{p_w\}$ do
12:     write lock $p$
13: end for
14: // update the begin timestamp
15: $tvv_{B(T)}[\,] = svv_i[\,]$
16: // Transaction Logic
17: Write uncommitted version of data items in partition $p_w$
18: Read the largest versioned data items $d$ in partition $p_r$,
    such that if $d$ is mastered at site $j$, $v(d) \leq tvv_B(T)[j]$
19: // Commit
20: // Update the commit version number
21: if wrote data items then
22:     increment $svv_i[i]$ and store as $v$
23: end if
24: // assign the commit timestamp
25: $tvv_T[] = tvv_{B(T)}[]$
26: $tvv_T[i] = v$
27: // Set the updated data items to version $v$
28: unlock partitions that were locked
29: // At the adaptation advisor
30: for each $S_j \in m$ sites do
31:     wait for $svv_j[]$ to reach $tvv_{B(T)}[\,]$
32:     write lock $S_j$ metadata
33:     if $S_j == S_i$ then
34:         $svv_j[j] = (max(tvv_T[j], svv_j[j])$
35:     end if
36:     $cvv_c[j] = tvv_T[j]$
37: end for
38: unlock $\{S_j\}$ metadata
39: return $cvv_c[\,]$
```

adaptation advisor initializes the begin timestamp to the latest site version number at each site (Line 6).

At the data site, the transaction waits for updates to be applied (Line 10), waiting for the freshness rules to be satisfied, then locks the partitions in the write set (Line 11) to ensure mutual exclusion of updates. As there may have been in-flight updates to the partition, the data site updates the begin timestamp (Line 15). Once the transaction has a begin timestamp, a transaction can execute its logic by writing new versioned data items (Line 17) and reading the data items belonging to the snapshot determined by the begin timestamp (Line 18).

When the transaction commits, if it updated data items it increments the site version number (Line 22). Then it copies that value as its commit timestamp and stamps the version on each newly created versioned data item (Line 27), which makes the updates visible. The adaptation advisor then updates its view of the site version vectors (Line 34) so that subsequent transactions abide by the commit order and avoid the long fork anomaly. Finally, the adaptation advisor updates the client version vector (Line 36) so that clients avoid transaction inversions within a session.

3.4.2 Adaptation Advisor

The adaptation advisor is responsible for routing transactions to sites for execution and deciding if, where and when to remaster data items using the strategies detailed in Chapter 3.3.

To make a remastering decision, the adaptation advisor must know the site that contains the master copy of the data item. To reduce the overhead of this metadata, the adaptation advisor supports grouping of data items into partitions[1], tracking master location on a per partition basis, and remastering data items in partition groups. For each partition group, DynaMast stores partition information that contains the current master location and a readers-writer lock.

To route a transaction, the adaptation advisor looks up the master location of each data item in the transaction's write set in a concurrent hash-table containing partition information. The adaptation advisor acquires each accessed partition's lock in shared read mode. If one site masters all partitions, then the adaptation advisor routes the transaction there and unlocks the partition information. Otherwise, the adaptation advisor makes a remastering decision and dynamically remasters the corresponding partitions to a single site. To do so, the adaptation advisor upgrades each partition information lock to exclusive write mode, which prevents concurrent remastering of a partition. Then, the adaptation advisor makes a remastering decision using vectorized operations that consider each site as a destination for remastering in parallel. Once the adaptation advisor chooses a destination site for the transaction, it remasters the necessary partitions using parallel `release` and `grant` operations, updates the master location in the partition and downgrades its lock to read mode. When the site masters the necessary partitions, the transaction begins executing, and the adaptation advisor releases all locks.

The adaptation advisor builds and maintains statistics such as data item access frequency and data item co-access likelihood for its strategies (Chapter 3.3) to effectively remaster data. Thus, the partition information also contains a counter that indicates the number of accesses to the partition and counts to track intra- and inter-transaction co-access frequencies. The adaptation advisor captures these statistics by adaptively sampling [29] transaction write sets and recording sampled transactions, and each transaction executed within a time window Δt (Equation 3.7) of it — submitted by the same client — in a transaction history queue. From these sampled write sets, the adaptation advisor determines partition level access and co-access frequencies, which it uses to make its remas-

[1]By default, the adaptation advisor groups sequential data items into equally sized partitions [191] though clients can supply their own grouping.

tering decisions. Finally, DynaMast expires samples from the transaction history queue by decrementing any associated access counts to adapt to changing workloads.

3.5 Performance Evaluation

I now present experimental results that show that DynaMast's ability to dynamically change data item mastership significantly improves system performance.

3.5.1 Evaluated Systems

To conduct an apples-to-apples comparison, I implemented all alternative designs within the DynaMast framework. Hence, all systems share the same data site, including the storage system, multi-version concurrency control scheme, and isolation level (strong-session snapshot isolation). This allows for the direct measurement and attribution of DynaMast's performance to the effectiveness of the dynamic mastering protocol and adaptation strategies.

DynaMast: I implemented the dynamic mastering protocol and adaptation strategies as the DynaMast system described in Chapter 3.4. In each experiment, DynaMast has no fixed initial data placement as it relies on its remastering strategies to distribute master copies among the sites.

Partition-Store: is the partitioned multi-master database system that I implemented in DynaMast. Partition-store uses table-specific partitioning (e.g. range, hash) to assign partitions to data sites, but does not replicate data except for static read-only tables. By using the offline tool Schism [53], I favoured partition-store to have superior partitioning for the OLTP workloads that I benchmark against. Partition-store uses the popular 2PC protocol to coordinate distributed transactions.

Multi-master: I implemented a replicated multi-master database system by augmenting partition-store to lazily maintain replicas. Thus, the multi-master system allows read-only transactions to execute at any site. As each data item has one master copy, updates to a data item occur only on the data item's master copy.

Single-Master: I leveraged DynaMast's adaptibility to design a single-master system in which all write transactions execute at a single (master) site while lazily maintaining read-only replicas at other sites. Single-master is superior to using a centralized system because the single-master system routes read-only transactions to (read-only) replicas, thereby reducing the load on the master.

LEAP: like DynaMast, guarantees single-site transaction execution but bases its architecture on a partitioned multi-master database without replication [114]. To guarantee single-site execution, LEAP localizes data in a transaction's read and write sets to the site where the transaction executes. To perform this data localization, LEAP does data shipping, copying data from the old master to the new master. I implemented LEAP by modifying the partition-store implementation.

3.5.2 Benchmark Workloads

In the experiments, each data site executes on a 12-core machine with 32 GB RAM. I also deployed an adaptation advisor machine and two machines that run Apache Kafka to ensure that there are enough resources available to provide timely and efficient update propagation. A 10Gbit/s network connects machines. All results are averages of at least five, 5-minute OLTPBench [58] runs with 95% confidence intervals shown as bars around the means.

Given the ubiquity of multi-data item transactions [58, 1] and workload access patterns [54, 114] present in a broad class of OLTP workloads, I incorporated these realistic characteristics into the **YCSB** workload. I used YCSB's scan transaction that reads from 200 to 1000 sequentially ordered keys, and enhanced the read-modify-write (RMW) transaction to update three keys. These modifications induce access correlations and multi-partition transactions, resulting in multi-site (distributed) transactions for multi-master and partition-store, remastering for DynaMast, and data-shipping for LEAP.

Each YCSB experiment uses four data sites containing an initial database size of 5 GB that grows to 30 GB of data by the end of the run, thereby taking up most of the available memory.

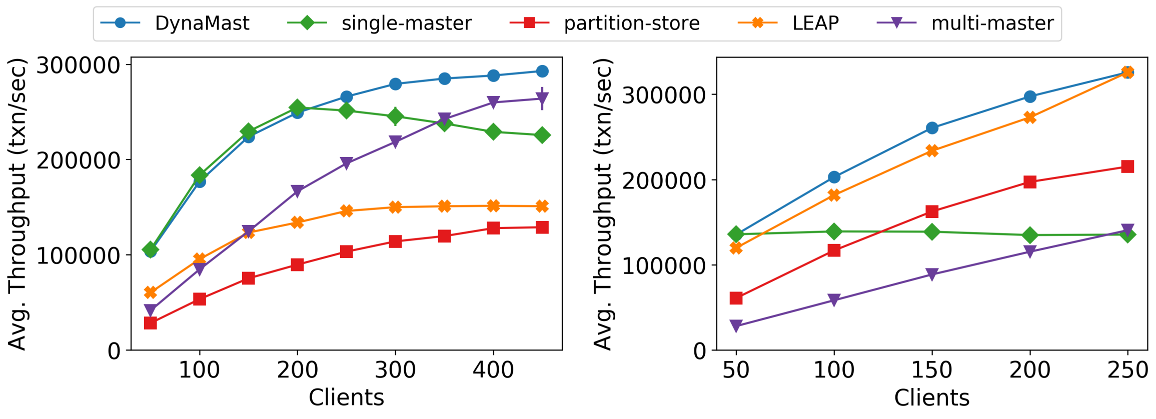
The **TPC-C** workload evaluation contains three transaction types: *New-Order*, *Payment* and *Stock-Level* that represent two update intensive transactions and a read-only transaction, respectively, and make up the bulk of both the workload and distributed transactions. By default, I use eight data sites, 350 concurrent clients and a 45% New-Order, 45% Payment, 10% Stock-Level mix that matches the default update and read-only transaction mix in the TPC-C benchmark. The TPC-C database has 10 warehouses and 100,000 items that grows to more than 20 GB of in-memory data per site by the end of an experiment run. Having more than this number of warehouses outstrips the physical memory of a data site machine.

The **SmallBank** workload models a banking application where users have checking and savings accounts and can transfer money between accounts, or check account balances. In the SmallBank transaction mix, there exist three types of transactions. First, single-row update transactions that account for 45% of the workload, including the *DepositChecking* transaction that adds money to a users' checking account. Second, two-row update transactions that comprise 40% of the workload mix, as in the *SendPayment* transaction, which atomically transfers money between two accounts. Third, the read-only *Balance* transaction that reads two rows and returns the sum of these rows — a users' checking and savings accounts — and occurs 15% of the time.

3.5.3 Results

I now present DynaMast's experimental results against partition-store, multi-master, single-master, and LEAP to demonstrate that remastering is efficient and effective.

Write-Intensive Workloads



(a) Uni. 50% RMW/50% Scan (b) Uni. 90% RMW/10% Scan

Figure 3.3: Experiment results for DynaMast in Write-Intensive YCSB Workloads.

Schism [53] reports that the partitioning strategy that minimizes the number of distributed transactions is range-partitioning. Thus, I assigned partition-store and multi-master a range-based partitioning scheme. DynaMast does not have a fixed partition placement and must learn access patterns to place partitions accordingly. As shown in

Figure 3.3a, DynaMast outperforms the other systems, improving transaction throughput by 2.3× over partition-store and 1.3× over single-master. Partition-store performs poorly compared to the other systems due to additional round-trips during transaction processing. While LEAP's transaction localization improves performance over partition-store by 20%, DynaMast delivers double the throughput of LEAP. DynaMast executes the scan operations at replicas without the need for remastering while LEAP incurs data transfer costs to localize both read-only and update transactions.

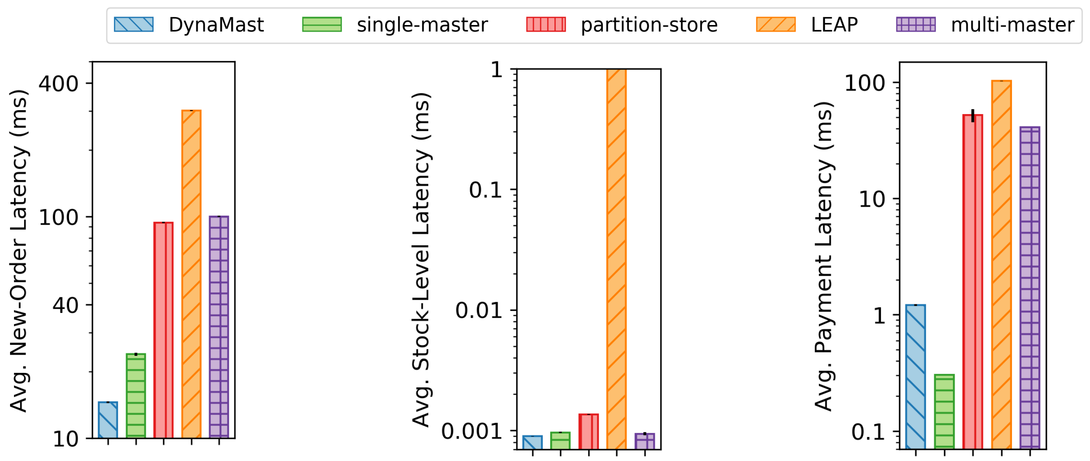
Single-master offloads scan transactions to replicas, thus avoiding distributed transactions without overloading the master site. However, as the number of clients increases in Figure 3.3a, the single-master site bottlenecks as it becomes saturated with update transactions. Like single-master, multi-master's replication allows scans to run at any site, which improves performance compared to partition-store. Multi-master avoids the single-site bottleneck as it distributes writes. However, its multi-site write transactions incur synchronization costs that DynaMast eliminates. Consequently, DynaMast's remastering strategies avoid the pitfalls of both single and multi-master; they ensure master copies of data are distributed evenly across the sites without executing distributed transactions, resulting in better resource utilization and thus superior performance.

Next, I increased the proportion of RMW transactions to 90% (Figure 3.3b), which increases the number of transactions that require remastering and increases contention. DynaMast continues to outperform the other systems by delivering almost 2.5× more throughput.

Complex Write Transactions

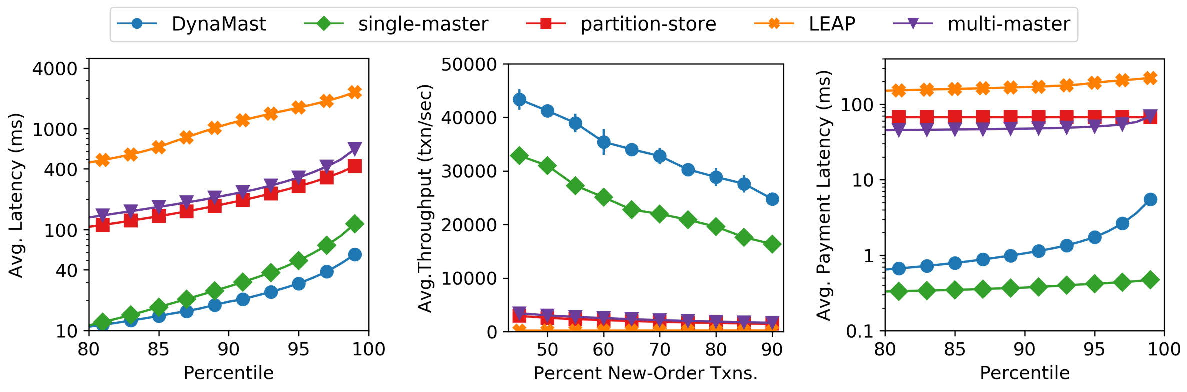
I studied the effect of a workload with more complex write transactions by using TPC-C. The New-Order transaction writes dozens of keys as part of its execution, increasing the challenge of efficiently and effectively placing data via remastering. TPC-C is not fully-partitionable due to cross-warehouse New-Order and Payment transactions but Schism confirms that the well-known partitioning by warehouse strategy minimizes the number of distributed transactions. Thus, I partitioned partition-store and multi-master by warehouse but force DynaMast to learn partition placements.

I first studied the differences in New-Order transaction latency among DynaMast and its comparators (Figure 3.4a). On average, DynaMast reduces the time taken to complete the New-Order transaction by a hefty 40% when compared to single-master. This large reduction in latency comes from DynaMast's ability to process New-Order transactions at all data sites, thereby spreading the load across the replicated system unlike that of single



(a) New-Order Lat. (b) Stock-Level Lat. (c) TPC-C Payment Lat.

Figure 3.4: Transactional Latency in the TPC-C workload.



(a) TPC-C Tail Lat. (b) TPC-C Throughput (c) Payment Tail Lat.

Figure 3.5: Tail latency and throughput for DynaMast in the TPC-C workload.

master. As shown in Figure 3.5a, the largest difference in transaction latency between DynaMast and single master exists in 10% of transactions that suffer the most from load effects. Specifically, DynaMast reduces the 90^{th} and 99^{th} percentile tail latency by 30% and 50% respectively compared to single-master.

DynaMast reduces average New-Order latency by 85% when compared to partition-store and multi-master, both of which perform similarly. DynaMast achieves this reduction by running the New-Order transaction at a single site and remastering to avoid the cost of multi-site transaction execution, which partition-store and multi-master must incur for every cross-warehouse transaction. Consequently, Figure 3.5a shows that both partition-store and multi-master have significantly higher (10×) 90^{th} percentile latencies compared to DynaMast.

DynaMast reduces New-Order latency by 96% over LEAP, which has no routing strategies and thus continually transfers data between sites to avoid distributed synchronization, unlike DynaMast's adaptive strategies that limit remastering. LEAP's localization efforts result in high transfer overheads and contention, manifesting in a 99^{th} percentile latency that is 40× higher than DynaMast's (Figure 3.5a).

DynaMast has low latency for the Stock-Level transaction (Figure 3.4b) because efficient update propagation rarely causes transactions to wait for updates, and multi-version concurrency control means that updates do not block read-only transactions. As single-master and multi-master also benefit from these optimizations, they have similar latency to DynaMast. Partition-store's average latency is higher because the Stock-Level transaction can depend on stock from multiple warehouses, necessitating a multi-site transaction. Although multi-site read-only transactions do not require distributed coordination, they are subject to straggler effects, increasing the probability of incurring higher latency as they must wait for all requests to complete. Thus, the slowest site's response time determines their performance. By contrast, LEAP, which lacks replicas, has orders of magnitude higher Stock-Level latency than DynaMast as it must localize read-only transactions.

Figure 3.5b shows how throughput varies with the percentage of New-Order transactions in the workload. When New-Order transactions dominate the workload, DynaMast delivers more than 15× the throughput of partition-store and multi-master, which suffer from multiple round trips and high tail latencies. Similarly, DynaMast delivers 20× the throughput of LEAP that lacks adaptive master transfer strategies and consequently continually moves data to avoid distributed coordination. DynaMast's significantly lower New-Order transaction latency results in throughput 1.64× that of single-master.

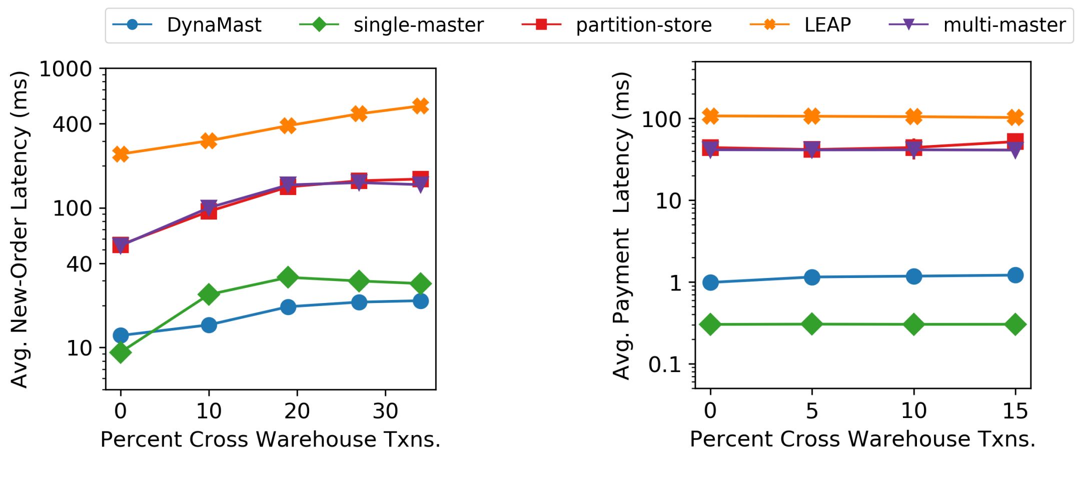
In the TPC-C benchmark, the *Payment* transaction is an update transaction, which records a payment by a customer. Similar to the New-Order transaction, 15% of the time, the Payment transaction updates a remote warehouse and district to simulate a customer paying for a remote order. However, unlike the New-Order transaction, the Payment transaction is much lighter as it updates only four rows by inserting a history of the payment and incrementing the payment totals for the relevant customer, district and warehouse.

Figures 3.4 and 3.5 present the experimental results for the Payment transaction for the TPC-C workload, with a 15% remote warehouse by default. As shown in Figure 3.4c, single-master has the lowest average latency at 0.3 ms, and DynaMast a mere 1.2 ms. However, as shown in Figure 3.5b DynaMast has higher overall throughput for the TPC-C workload overall. Consequently, DynaMast trades off a small increase in Payment transaction latency for improvements in the workload overall.

There are two primary causes for this trade off in performance. First, routing all Payment transactions to a single-master does not place a heavy load on this node, when compared to the New-Order transaction. Hence, the single-master does not suffer from load effects. Second, as the workload is not perfectly partitionable, DynaMast must perform some remastering to execute transactions at a single site, an operation not necessary for single-master. Figure 3.5c highlights the cost of this remastering on the Payment transaction, as DynaMast only differs from single-master significantly in the slowest 10% of transactions. Although DynaMast could master all data items at a single-node, doing so would significantly increase the latency of the New-Order transaction, as we show next.

As with the New-Order transaction, observe that LEAP, partition-store and multi-master experience orders of magnitude higher transaction latency for the Payment transaction, and DynaMast reduces Payment latency by 99%, 97% and 96% over these competitors, respectively (Figure 3.4c).

Decreasing Transaction Access Locality



(a) New-Order Lat.

(b) Payment Lat.

Figure 3.6: Effects of Varying Cross-Warehouse Transactions in DynaMast

As shown in Figure 3.6a, DynaMast reduces New-Order transaction latency by an average of 87% over partition-store and multi-master when one-third of New-Order transactions cross warehouses. Partition-store and multi-master's New-Order latency increases almost 3× over having no cross warehouse transactions to when one-third of the transactions cross warehouses, which results in an increase of only 1.75× in DynaMast. LEAP increases the New-Order latency by more than 2.2× as more distributed transactions necessitate more data transfers while DynaMast brings its design advantages that include routing strategies and remastering to bear, delivering better performance than LEAP.

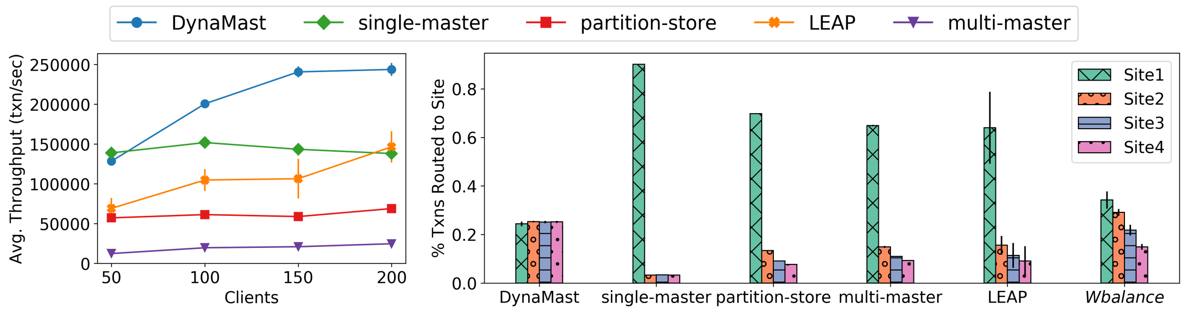
Partition-store and multi-master's latency increase because cross warehouse transactions also slow down single warehouse transactions [94]. Recall that 2PC requires holding locks during the uncertain phase to prevent uncommitted changes from being visible to other transactions, which therefore also blocks local transactions from executing. As the number of cross-warehouse transactions increases, DynaMast recognizes that being a more dominant single-master system can bring performance benefits, and therefore reacts by mastering more data items at one site. However, DynaMast knowingly avoids routing all New-Order transactions to a single site to avoid placing excessive load on it. These techniques allow DynaMast to significantly reduce New-Order latency by 25% over single-master.

Figure 3.6b presents the average latency of the payment transaction as the rate of cross warehouse Payment transactions increase. Observe that latency increases just a mere 0.2 ms for DynaMast whereas latency for partition-store and multi-master increase by nearly 10 ms as the number of cross warehouse Payment transactions increases from 0 to the default 15%. As in the experiment with cross warehouse New-Order transactions, this experiment demonstrates that DynaMast adds little overhead to transaction execution, and has effective master placement strategies when compared to partition-store. Finally, observe that single-master experiences little change in Payment transaction latency as the number of cross-warehouse transactions increase because the lightweight transactions do not increase contention as was the case for the New-Order transaction.

Skewed Workloads

I evaluated DynaMast's ability to balance load in the presence of skew via remastering with a YCSB-based Zipfian 90%/10% RMW/scan workload (Figure 3.7a).

DynaMast significantly outperforms its comparators, improving throughput over multi-master by 10×, partition-store by 4×, single-master by 1.8×, and LEAP by 1.6×. Partition-store's performance suffers as it cannot distribute heavily-accessed partitions to multiple



(a) Skew 90% RMW/10% Scan (b) Skew Transaction Routing

Figure 3.7: Effects of Skew on DynaMast in YCSB

sites (Figure 3.7b), resulting in increased transaction latency due to resource contention. Multi-master suffers the same fate while additionally having to propagate updates, which further degrades performance. LEAP shows better throughput than partition-store as it executes transactions at one site but suffers from co-location of hot (skewed) partitions. Resource contention degrades performance for single-master as all update transactions must execute at the single master site. DynaMast mitigates the performance issues of its competitors by spreading updates to master data partitions over all sites in the replicated system evenly, resulting in balanced load and superior performance.

Learning Changing Workloads

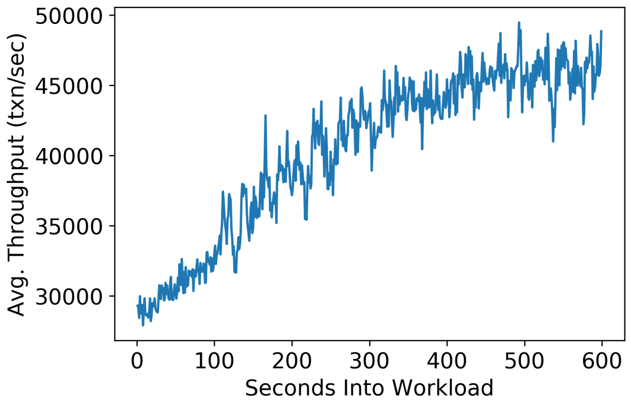


Figure 3.8: DynaMast's Adaptivity Over Time

Access patterns in workloads often change [191] resulting in degraded performance when data mastership/allocation is fixed. A key feature of DynaMast is its ability to adapt to these workload changes by localizing transactions.

To demonstrate this adaptive capability, I conducted a YCSB experiment with mastership allocated manually using range partitioning but with the workload utilizing randomized partition access. Hence, DynaMast must learn new correlations and remaster data to maintain performance.

I deployed 100 clients running 100% RMW transactions accessing data with skew. This challenges DynaMast with *high contention* and the remastering of *nearly every* data item. As Figure 3.8 shows, DynaMast rises to this challenge, continuously improving performance over the measurement interval resulting in a throughput increase of 1.6× from when the workload change was initiated. This improvement showcases DynaMast's ability to learn new relationships between data partitions and its strategies, leveraging this information effectively to localize transactions via remastering.

Remastering Analysis

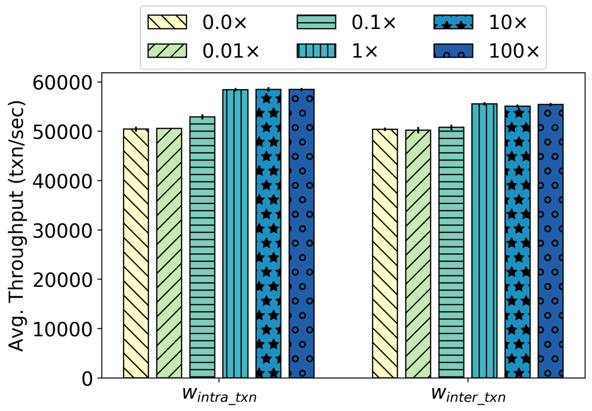


Figure 3.9: Co-Access Sensitivity in DynaMast

Recall that DynaMast makes remastering decisions by employing a linear model (Equation 3.8) that contains four hyperparameters ($w_{balance}$, w_{delay}, w_{intra_txn}, w_{inter_txn}). To determine the effects of these parameters on the adaptation advisor's master placement decisions and subsequently on performance, I performed sensitivity experiments using a skewed YCSB workload. I varied each parameter from its default value (normalized to 1) by scaling one or two orders of magnitude up, and down. I also set each parameter, in turn, to 0 to determine the effect of removing the feature associated with that parameter from the adaptation advisor's strategy.

When every parameter is non-zero, throughput remains at 8% of the maximal throughput, demonstrating DynaMast's robustness to variation in parameter values.

When $w_{balance}$ is 0, throughput drops by nearly 40% as DynaMast increasingly masters data at one master site since no other feature encourages load balance. Figure 3.7b shows the effects on transaction routing when $w_{balance}$ is scaled to 0.01 of its default: 34% of the requests go to the most frequently accessed site while 13% of requests go to the least frequently accessed site, compared to even routing (25%) by default.

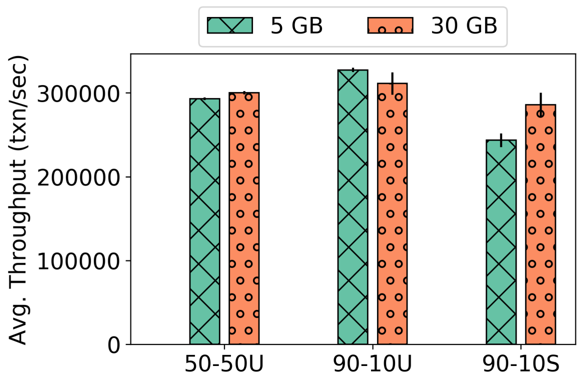
The w_{intra_txn} and w_{inter_txn} hyperparameters complement each other; when one is 0, the other promotes co-location of co-accessed data items. To further examine these effects, I induce workload change, as before, so that learning and understanding data item access patterns is of utmost importance.

Figure 3.9 shows throughput increasing as w_{intra_txn} increases from 0 to the relative value of 1 used in experiments. This 16% improvement in throughput demonstrates that the adaptation advisor captures the intra-transactional co-access patterns, using them to co-locate the master copies of co-accessed data items. Observe that a similar trend of throughput increasing by 10% occurs when the inter-transactional co-access parameter (w_{inter_txn}) varies.

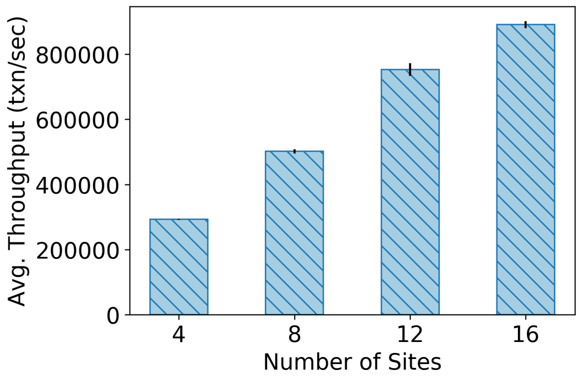
Scalability Results

To show that DynaMast can support larger database sizes, I evaluated its performance on the YCSB workloads using an initial database size of 30 GB. Over the course of the experiment, the database size grows to 120 GB given that the system keeps at least 4 versions of each record. Consequently, I added memory to the database machines to bring them to 128 GB of RAM. No other aspects of the system were changed for these experiments.

Figure 3.10a shows DynaMast's throughput for the YCSB workloads with initial database sizes of 5 GB and 30 GB that grow to occupy nearly all of the memory of the data site machines. Observe that there is little variation in performance as the database size increases for the uniform 50/50 (50-50U) and 90/10 RMW/Scan workloads (90-10U), though DynaMast does experience a slight performance degradation on the write-intensive workload due to increased data tracking and management overheads at the adaptation advisor and increased remastering. DynaMast's performance on the skewed workload (90-10S) increases



(a) Data Size Scalability



(b) Scaling Across Data Sites

Figure 3.10: DynaMast's Scalability

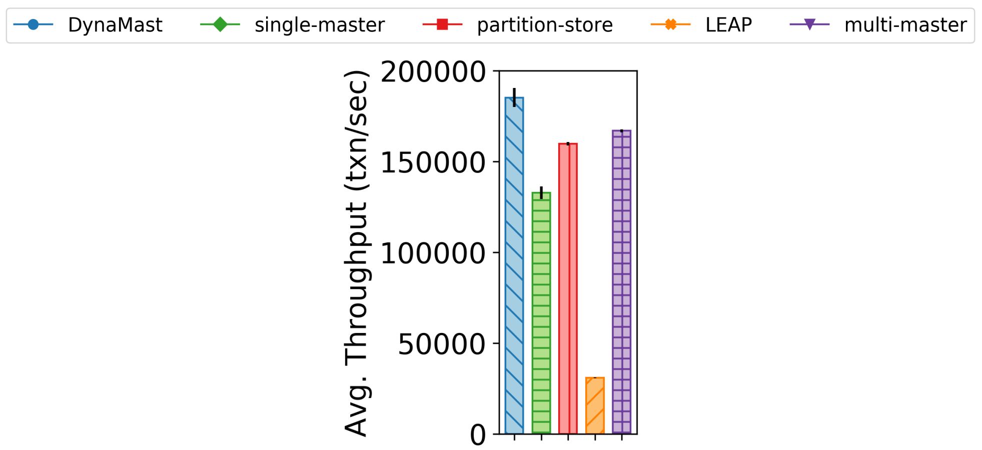
because the access skew is spread across more items, and therefore decreases contention. These experiments show that DynaMast's adaptive remastering strategies and underlying infrastructure continue to perform well as database size grows.

To assess DynaMast's ability to scale across an increasing number of data sites, I evaluated DynaMast using 4, 8, 12 and 16 data sites using the uniform YCSB workload with a 50% RMW and 50% scan mix. I used a balanced read-write workload and a 5 GB database size to reduce write contention on individual partitions and attribute DynaMast's scaling capabilities to effective resource utilization.

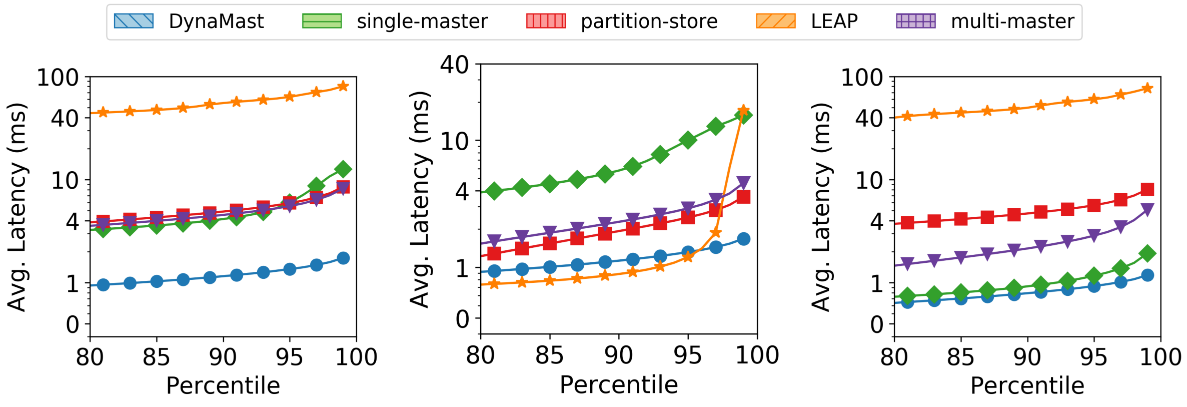
Figure 3.10b shows a maximum throughput comparison for DynaMast as the number of data sites increase. Observe that DynaMast improves throughput by more than 3× as the number of sites grows by a factor of 4. DynaMast achieves this near-linear scalability because it can effectively distribute requests among sites and therefore leverage their resources to improve performance. As the number of sites increases, the rate of increase in throughput slows, a consequence of maintaining full replicas at each data site by applying transactional updates. Finally, even as nearly 900,000 transactions per second proceed through the system, the adaptation advisor is not a bottleneck.

Effect of Short Transactions

To stress the transaction protocol, I next evaluate DynaMast using a workload that contains short transactions. In this workload, unlike TPC-C and YCSB, transactions access at most two records, which are the minimum necessary for different sites to master data accessed in



(a) SmallBank Throughput



(b) Two-Row Update (c) Single-Row Update (d) Two-Row Read

Figure 3.11: Experimental results for the SmallBank workload showing maximum throughput and the tail latency for SmallBank's three transaction classses.

the transaction, and trigger remastering in DynaMast, 2PC in partition-store and multi-master, or data shipping in LEAP. Such a workload places a different burden on systems than the heavier TPC-C transactions as the underlying transaction protocol dominates transaction execution time, not the actual transaction logic. To do so, I use the SmallBank workload, which models a banking application.

As shown in Figure 3.11a, DynaMast has the highest throughput in the SmallBank workload, when compared to partition-store (by 15%), multi-master (by 10%), single-master (by 40%) and LEAP (more than 600%). To understand DynaMast's improvement in throughput, I examined the distribution of transaction latencies for the three transaction types within the workload, and present their tail latencies in Figures 3.11b, 3.11c and 3.11d. Comparing DynaMast with single-master, observe that the load effect of routing all updates to a single-site is more than 7× higher tail latencies for update transactions (Figures 3.11b and 3.11c), whereas DynaMast dissipates the update load among sites. By contrast, read-only transactions (Figure 3.11d) run at replicas for single-master and therefore have similar latencies to DynaMast.

LEAP initially has slightly lower single-row update latencies than DynaMast (Figure 3.11c), as DynaMast requires system resources to maintain replicas asynchronously. However, LEAP suffers from high tail latencies for these single-row transactions, as they must wait for data migration that is necessary for multi-row transactions to complete. As LEAP does not have smart routing strategies, LEAP suffers from frequent and expensive data migration, which increases multi-row transaction latency by nearly 40 × that of DynaMast (Figures 3.11b and 3.11d). As with LEAP, partition-store initially has a similar single-row transaction latency to DynaMast (Figure 3.11c); however, the requirements of the uncertain phase during distributed transaction processing force blocking — even for single-row transactions — which increases tail latency. At the tail of multi-row transactions (Figures 3.11b and 3.11d), which require the expensive two-phase commit for partition-store, observe that DynaMast has latency that is a quarter of partition-store. Multi-master exhibits similar trends to partition-store for update transactions, as both systems incur distributed transactions, however, multi-master must propagate updates which increases tail latency slightly when compared to partition-store. For read transactions, multi-master can leverage the existence of replicas and execute at a single site, which reduces transaction latency to levels closer to that of single-master and DynaMast. This reduction in read-transaction latency compared to partition-store contributes to multi-master's higher throughput.

In summary, DynaMast significantly reduces the tail latency of transactions in SmallBank, thereby demonstrating the benefits of the dynamic mastering protocol and the transaction routing strategies.

Performance Breakdown

DynaMast is designed to be a system that delivers significant performance benefits with low overhead. The latencies for routing decisions, wait time for pending updates and transaction commit time are low. DynaMast's comprehensive strategies that learn access correlations are effective at minimizing remastering as less than 3% of transactions require remastering.

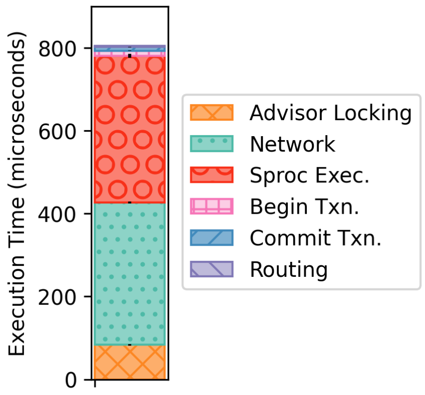


Figure 3.12: A breakdown of transactional latencies in DynaMast.

Figure 3.12 plots a breakdown of DynaMast's average transaction execution time (Figure 3.12) during a Uniform 50/50 RMW/Scan YCSB experiment. I divide latency into six categories: the time that it takes the adaptation advisor to lock and identify the location of data items, which accounts for 10% of the overall average response time; the time taken to make a routing decision — including remastering — that takes less than 1% of the overall time as DynaMast amortizes the cost of remastering across many transactions; the amount of time that requests spend in the network between system components, which is more than 40% of the time; the actual execution time of the database stored procedure (actual transaction logic) accounts for 45% of overall latency; the time to begin a transactions, including lock acquisition at the data site and waiting for any session state, which takes less than 1% of the overall time; and the time to commit a transaction that takes just over 1% of the overall time.

Transaction routing accounts for less than 1% of the overall transaction time due to the amortization of remastering across many transactions. DynaMast achieves this low overhead because its strategies aim to minimize future remastering by modelling inter- and intra-transactional data access correlations. Consequently, less than 1% of transactions in the YCSB and SmallBank workloads and less than 3% in TPC-C require remastering. I additionally measured the network overhead of remastering and DynaMast as a whole. In a YCSB workload that generated an average of 43 MB/s of stored procedure arguments, propagating refresh transactions consumed 155 MB/s of network traffic — traffic necessary in any replicated system. Remastering requests accounted for a meager 3 MB/s of network traffic. These results indicate that DynaMast adds minimal overhead to transaction execution, while significantly outperforming the other systems for different workload characteristics.

3.6 Summary

In this Chapter, I presented DynaMast, a multi-master replicated database system that guarantees one-site transaction execution through dynamic mastering. As shown experimentally, DynaMast's novel remastering protocol and adaptive strategies are lightweight, ensure balanced load among sites, and minimize remastering as part of future transactions. Consequently, DynaMast eschews multi-master's expensive distributed coordination and avoids the single-master site bottleneck. These design strengths allow DynaMast to improve performance by up to 15$\times$ over prior replicated system designs.

Chapter 4

MorphoSys: Automatic Physical Design Metamorphosis for Distributed Database Systems

Recall from Chapter 1, that distributed DBMSs use data replication and partitioning to distribute data among sites. The data replication and partitioning schemes, chosen by an administrator or system, form a distributed database's *physical design*. A distributed physical design must decide (i) what the data partitions should be, (ii) which data partitions to replicate, and (iii) where (at which site) to place the master (or updateable) copy of a partition, and where to place any replicas (or secondary) copies of partitions. These decisions, in turn, determine the sites where each transaction executes. Consequently, *how* data is stored (partitioning and replication) and *where* (location of master and replica copies) influences transaction execution.

In this chapter, I present how **MorphoSys** *automatically* decides how to partition data, what to replicate, and where to place these partitioned and replicated data. MorphoSys learns from workload locality characteristics what the partitions should be, and where to place master and replica partitions, *dynamically*. This dynamism frees the system from having to know workload access patterns a priori, allowing MorphoSys to *change* or metamorphosize the distributed physical design on-the-fly. Moreover, to avoid expensive multi-site transaction coordination, MorphoSys dynamically alters its physical design to co-locate co-accessed data and guarantee single-site transaction execution via dynamic replication and the remastering protocol described in Chapter 3.

Remarkably, once MorphoSys starts executing, it requires no administrator intervention. Unlike prior approaches [172, 201], MorphoSys adapts its physical design continuously and iteratively, adjusting *both* its replication and partitioning schemes to cater to the workload. MorphoSys uses a *learned* cost model based on workload observations to decide how, and when, to alter its physical design.

4.1 System Model

For every partition, MorphoSys decides the site that stores the master copy of the partition and the sites that lazily replicate the partition. A change in any of these decisions results in a physical design change. MorphoSys acts on these decisions, using physical design change operations (Chapter 4.2) to dynamically change the system's data replication, partitioning and mastering schemes.

A data partition is defined a contiguous range of data items based on their *row id*'s. Partition p contains all data items with *row id*'s in an inclusive row range between the partition start and end, denoted as $(start(p), end(p))$. A partition is mastered at a single site and has replicas on a possibly empty set of sites.

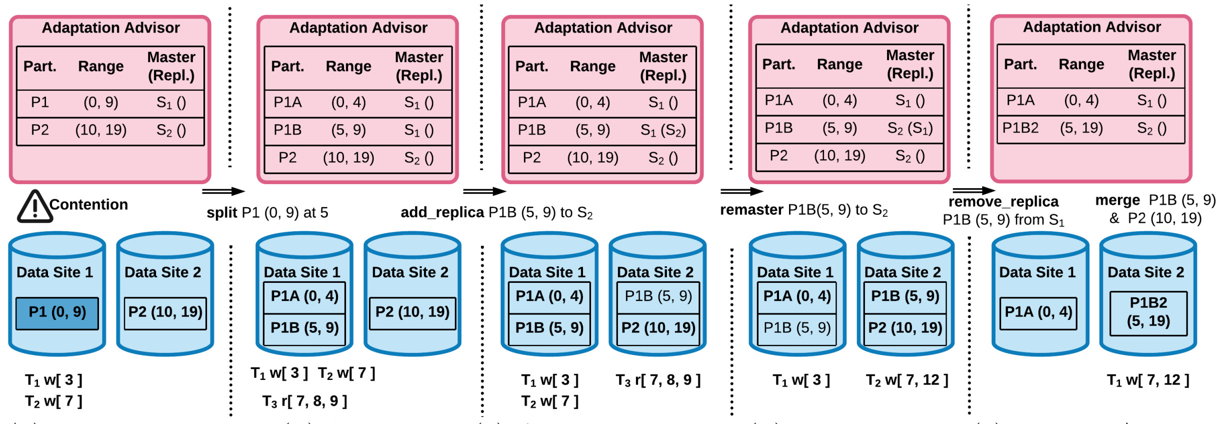
4.2 Physical Design Change Operations

MorphoSys supports a variety of dynamic physical design change operations. These operations are flexible building-blocks that MorphoSys effectively combines to produce efficient distributed database physical designs.

Figure 4.1 exemplifies MorphoSys' physical design change operations by example, using two data sites. Figure 4.1a shows the initial design, while Figures 4.1b–4.1e illustrates the series of physical design changes that culminate in a new physical design.

In the initial physical design (Figure 4.1a), transaction T_1 writes data item 3, and transaction T_2 writes data item 7, hence both update the partition $(0, 9)$ located on data site 1. Thus, there is physical lock contention on the partition even though the updates logically do not conflict. *Splitting* the partition, a dynamic partitioning operation, changes the data partition that data items belong to and dissipates the contention (Figure 4.1b) as transactions T_1 and T_2 subsequently update disjoint partitions.

In Figure 4.1c, three transactions execute on data site 1, including the read-only transaction T_3, adding load to the site. As replicas service read-only transactions, dynamically



(a) Initial State (b) Split (c) Add Replica (d) Remaster (e) Remove/Merge

Figure 4.1: An overview of physical design change operators in MorphoSys, and their effects.

replicating to *add a replica* of partition $(5, 9)$ to data site 2 allows T_3 to execute there, thereby distributing load among sites.

In Figure 4.1d transaction T_2 updates data items 7 and 12. To guarantee single-site transaction execution, MorphoSys ensures that one data site contains the master copies of the partitions containing data items 7 and 12 — partitions $(5, 9)$ and $(10, 19)$ — by dynamically changing the location of the master of partition $(5, 9)$ from data site 1 to data site 2 via *remastering*. Single-site transaction execution ensures that transactions are not blocked waiting for distributed state during 2PC's uncertain phase that increases lock holding time while blocking local and distributed transactions [94, 76, 12, 119]. Observe that data site 2 was previously replicating partition $(5, 9)$, which ensures remastering is efficient [119, 12].

Replicas require space for storage and their maintenance consumes resources. MorphoSys dynamically *removes replicas* of partitions if there is little benefit in maintaining a replica, such as when no transactions read from the replica, as in Figure 4.1e for partition $(5, 9)$ at data site 1.

Finally, to reduce the metadata overhead of tracking partitions, MorphoSys *merges* co-accessed partitions together, as shown in Figure 4.1e for partitions $(5, 9)$ and $(10, 19)$ creating partition $(5, 19)$.

The examples in Figure 4.1 show the benefits of physical design changes. MorphoSys performs changes prudently by considering the workload to avoid unnecessary changes or changes that it will undo in the immediate future.

4.2.1 Formal Definitions of Physical Design Change Operators

I have previously outlined the definition of a partition, the requirements for transactions, and the five physical design change operators. I now formalize these definitions.

A partition of data p contains all data items with *row id*'s that fall in the inclusive range $(\mathit{start}(p), \mathit{end}(p))$. The partition p has its master copy located at site $S_i = \mathit{master}(p)$, and replicas placed at a (possibly empty) set of sites $\{S_j | j \neq i\} = \mathit{replicas}(p)$.

I formally define the data items in a transaction's read set as $\{d_r\}$, and the data items in its write set as $\{d_w\}$. Given these data items, the adaptation advisor identifies the set of partitions $\{p_r\}$ and $\{p_w\}$ as the read and write set, respectively. To identify a partition given a data item d, the adaptation advisor finds the partition p such that $\mathit{start}(p) \leq d \leq \mathit{end}(p)$ holds. A transaction can execute at a site S if for all p_w in the write set $\mathit{master}(p_w) = S$ holds, and if for all p_r in the read set $S \in \{\mathit{master}(p_r)\} \cup \mathit{replicas}(p_r)$ holds

The definition of the five physical design change operators is as follows.

split: given a partition p, and k such that $\mathit{start}(p) < k \leq \mathit{end}(p)$, then $\mathit{split}(p, k)$, creates new partitions p_L and p_H and removes p. Partitions p_L and p_H are defined such that $p_L = (\mathit{start}(p), k - 1)$ and $p_H = (k, \mathit{end}(p))$, $\mathit{master}(p_L) = \mathit{master}(p_H) = \mathit{master}(p)$, and $\mathit{replicas}(p_L) = \mathit{replicas}(p_H) = \mathit{replicas}(p)$.

merge: given partitions p_L, and p_H then $\mathit{merge}(p_L, p_H)$, creates new partition p, and removes p_L and p_H. For *merge* to succeed, p_L and p_H must satisfy the following $\mathit{end}(p_L) = \mathit{start}(p_H) - 1$, $\mathit{master}(p_L) = \mathit{master}(p_H)$, and $\mathit{replicas}(p_L) = \mathit{replicas}(p_H)$. Partition p is defined such that $p = (\mathit{start}(p_L), \mathit{end}(p_H))$, $\mathit{master}(p) = \mathit{master}(p_L) = \mathit{master}(p_H)$, and set $\mathit{replicas}(p) = \mathit{replicas}(p_L) = \mathit{replicas}(p_H)$.

add replica: given partition p, with $\mathit{replicas}(p) = R$, and S_j, such that $S_j \neq \mathit{master}(p)$ and $S_j \notin R$ then
$\mathit{add\_replica}(p, S_j)$ sets $\mathit{replicas}(p) = R \cup \{S_j\}$.

remove replica: given partition p, with $\mathit{replicas}(p) = R$, and S_j, such that $S_j \in R$ then $\mathit{remove\_replica}(p, S_j)$ sets $\mathit{replicas}(p) = R \setminus \{S_j\}$.

remaster: given partition p, with $\mathit{replicas}(p) = R$, and S_j, such that $S_j \in R$, and $\mathit{master}(p) = S_i$, then $\mathit{remaster}(p, S_j)$, sets $\mathit{master}(p) = S_j$, and sets $\mathit{replicas}(p) = R \setminus \{S_j\} \cup \{S_i\}$.

4.3 Transaction Execution and Physical Design Change

I now describe MorphoSys' novel concurrency control and update propagation mechanisms and how they provide efficient physical design changes and transaction execution. I begin by presenting the design to achieve efficient transaction execution followed by the mechanics of the physical design operators.

4.3.1 Transaction Isolation and Concurrency

MorphoSys *decouples read and write operations* as these operations do not conflict under SSSI. This decoupling allows concurrent execution of reads at replicas while data sites apply propagated updates. MorphoSys uses multi-versioning [105, 57] to efficiently decouple reads and writes.

All of MorphoSys' operations, including transactions and physical design change operations, occur on a *per partition* basis. That is, operations access a partition or its metadata only if accessed in a transaction's read or write set or as part of a design change operator. Per partition operations minimize contention and blocking as the system accesses only relevant partitions, and replicas wait for only the necessary updates to relevant partitions to preserve consistency.

4.3.2 Partition-Based Multi-version Concurrency Control

Guided by the design requirements of supporting session consistency with SI, decoupling reads and writes, and performing operations on a per partition basis, MorphoSys uses a novel concurrency control algorithm. The algorithm maintains per partition version information and a lock per partition. Any update to a partition by transactions or physical design change operators acquires the partition's lock using lock ordering to avoid deadlocks. Consequently, at most one writer to a partition executes at a time, which prevents unnecessary transactional aborts. Concurrent writes and design changes to different partitions are supported since these are conflict-free with locks held on a per partition basis. Dynamic partitioning, through *split* and *merge* operators, ameliorates contention within and across partitions. Read operations leverage multi-versioning to execute freely without acquiring partition locks.

MorphoSys uses per partition version information and tracks transactional dependencies to implement a dependency-based concurrency control protocol [147, 146], which allows

MorphoSys to apply updates in parallel and on a per partition basis at replicas (Chapter 4.3.3). To track partition dependencies, each partition p maintains a *version number* $v(p)$. Multiple versions of each data item d are kept, and each version is called a *versioned data item*, consisting of a version number $v(d)$ that coincides with the version number $v(p)$ of the partition p that contains d when d was updated. When a transaction T updates data item d in a partition p, it creates a new version of the data item and its associated data. On commit, T increments the partition version number $v(p)$ and assigns that number as $v(d)$ for the new versioned data item.

In addition to storing per partition version numbers, MorphoSys maintains dependency information to ensure transactions read from a consistent snapshot of data. In particular, for version $v(p_i)$ in partition p_i, MorphoSys stores the version number of partition p_j belonging to the same logical snapshot as $v(p_i)$, which I denote as $\mathit{depends}(p_i, v(p_i), p_j)$.

To track dependencies, MorphoSys uses the following *dependency recording rule*. If a transaction T updates data items in partitions p_i and p_j, MorphoSys assigns partition version numbers $v(p_i)$ and $v(p_j)$ when T commits. MorphoSys then records $v(p_j)$ as $\mathit{depends}(p_i, v(p_i), p_j)$, and similarly $v(p_i)$ as $\mathit{depends}(p_j, v(p_j), p_i)$. To capture any existing transitive dependencies, MorphoSys extends the recording rule to the rule presented in Equation 4.1, which applies to any partition p_k that MorphoSys has not already recorded dependencies for as part of the transaction update, and partitions p_i and p_j in the write set. Equation 4.1 captures direct dependencies that involve partitions updated in the same transaction, and transitive dependencies that are inherited from previous transactions. Taking the maximum of a partition's direct and inherited dependencies ensures transactions observe consistent state.

$$\mathit{depends}(p_i, v(p_i), p_k) = \mathit{max}_j(\mathit{depends}(p_i, v(p_i) - 1, p_k), \mathit{depends}(p_j, v(p_j) - 1, p_k)) \quad (4.1)$$

MorphoSys efficiently maintains tracked dependencies by storing them in recency order with partition metadata at a data site. As OLTP transactions typically access a small amount of data relative to the database size [113, 58], tracked dependencies are often small, while long-running transactions accessing many partitions induce more dependencies for the system to track. To ensure tracked dependencies do not grow unbounded, MorphoSys garbage collects versioned data items and dependency information of partitions with version numbers lower than a watermark version number, which is the smallest most recent version number at any site.

Strong Session Snapshot Isolation

To enforce SSSI, MorphoSys uses the recorded dependencies to guarantee that transactions read data from a consistent snapshot state using the *consistent read rule*: a transaction T reads versions $v(p_i)$ of every partition p in its read set such that $v(p_i) \geq \mathit{depends}(p_j, v(p_j), p_i)$ if p_j is also in T's read set. T reads the latest version of data item d such that $v(d) \leq v(p_i)$.

MorphoSys uses the *depends* relationship as the basis of transaction timestamps. When a transaction T intends to read partitions $\{p_r\}$ and write partitions $\{p_w\}$, T acquires write locks on each p_w. T then reads the version number for each partition $v(p)$ in its read and write set, and stores this value as $T^B(p)$. As defined by the consistent read rule, T then determines the version $T^B(p_i)$ of each partition such that $T^B(p_i) \geq \mathit{depends}(p_j, T^B(p_j), p_i)$ for p_i and p_j in T's read and write set. The values $\{p, T^B(p)\}$ form T's begin timestamp T^B. T then reads and writes transactionally.

On commit, MorphoSys generates a commit timestamp T^C by incrementing $v(p_w)$ for each p_w in T's write set and follows the dependency recording rule. Mutual exclusion of partition writes ensures that the updated version $v(p_w) = T^B(p_w) + 1$, which is assigned to $T^C(p_w)$, and along with $T^B(p_r)$ for each read partition forms the commit timestamp.

Formalizing MorphoSys' Concurrency Control Protocol

Given the previous description of different aspects of MorphoSys' concurrency control protocol, I now describe the complete algorithm. I first explain how MorphoSys modifies transaction timestamps to eliminate transaction inversions and long fork anomalies. Then I present the complete concurrency control algorithm.

To prevent transaction inversion, MorphoSys maintains per client session state called a session timestamp. A client, C, maintains session state $C^S = \{p, \max_T T^C(p)\}$ for all transactions T submitted by the client, which represents the latest version number of all partitions it has accessed. MorphoSys initially sets $T^B(p) = C^S(p)$ for every partition p in T's read and write set. Hence, when T executes, it observes the state at least as up-to-date as the last observed state.

MorphoSys like DynaMast, uses the adaptation advisor to enforce the total commit order and avoid the long fork anomaly. However, in MorphoSys the adaptation advisor tracks the *committed version numbers* of partitions.

MorphoSys' concurrency control algorithm is presented in Algorithm 3. MorphoSys initializes a transaction's begin and commit timestamp to the session timestamp to ensure

Algorithm 3 MorphoSys Concurrency Control Protocol

```
Require: Transaction $T$'s sorted write set $\{p_w\}$, read set $\{p_r\}$, and session timestamp, $C^S$
Ensure: The updated session timestamp
1: // Begin
2: // At the adaptation advisor
3: $T^B = C^S$
4: $\{p\} = $ sort $\{p_w\} \cup \{p_r\}$
5: for $p \in \{p\}$ do
6:     read lock $p$ metadata
7:     $T^B(p) = p.commit\_version\_number$
8: end for
9: unlock $\{p\}$ partition metadata
10: // At the data site
11: for $p \in \{p\}$ do
12:     wait for $p$'s version to reach $T^B(p)$
13:     if $p \in \{p_w\}$ then
14:         write lock $p$
15:         $T^B(p) = p$'s version number
16:     end if
17: end for
18: $partitions\_to\_check = \{p\}$
19: for $p_i \in partitions\_to\_check$ do
20:     remove $p_i$ from $partitions\_to\_check$
21:     for $p_j \in \{p\}$ do
22:         if $T^B(p_i) < depends(p_j, T^B(p_j), p_i)$ then
23:             $T^B(p_i) = depends(p_j, T^B(p_j), p_i)$
24:             $T^C(p_i) = T^B(p_i)$
25:             wait for $p$'s version to reach $T^B(p)$
26:             insert $p_i$ into $partitions\_to\_check$
27:         end if
28:     end for
29: end for
30: $T^C = T^B$
31: for $p \in \{p_w\}$ do
32:     $T^C(p) = T^B(p) + 1$
33: end for
34: // Transaction Logic
35: Write versioned data items in partition $p_w$ with version $T^C(p_w)$
36: Read the largest versioned data items $d$ in partition $p_r$, such that $v(d) \leq T^B(p_r)$
37: // Commit
38: $loc\_depends = T^C$
39: for $p_i \in \{p\}$ do
40:     for $p_j \in depends(p_i, T^B(p_i))$ do
41:         if $depends(p_i, T^B(p_i), p_j) > loc\_depends(p_j)$ then
42:             $loc\_depends(p_j) = depends(p_i, T^B(p_i), p_j)$
43:         end if
44:     end for
45: end for
46: for $p \in \{p_w\}$ do
47:     $depends(p_i, T^C(p_i)) = loc\_depends$
48:     set $p$'s version to $T^C(p)$
49:     unlock $p$
50: end for
51: // At the adaptation advisor
52: for $p \in \{p\}$ do
53:     if $p \in \{p_w\}$ then
54:         wait for $p.commit\_version\_number$ to reach $T^B(p)$
55:         write lock $p$ metadata
56:         $p.commit\_version\_number = T^C(p)$
57:     end if
58:     $C^S(p) = \mathbf{max}(T^C(p), C^S(p))$
59: end for
60: unlock $\{p_w\}$ partition metadata
61: return $C^S$
```

the transaction observes state at least as up to date as the latest previously observed state from the same session (Line 3). To ensure transactions respect a total commit order, the adaptation advisor initializes the begin timestamps to the *commit version number* (Line 6). At the data site, the transaction locks the partitions in the write set (Line 13) to ensure mutual exclusion of updates. As there may have been in-flight updates to the partition, the data site updates the begin timestamp for partitions in the write set (Line 15). Then, the data site ensure the transaction observes a consistent snapshot by following the *consistent read rule* (Lines 22 and 23). Once the transaction has a consistent begin timestamp, MorphoSys constructs the commit timestamp (Lines 30 to 32) by copying the begin timestamp for read partitions, and incrementing the begin timestamp for write partitions. Once begin and commit timestamps are assigned, a transaction can execute its logic by writing new versioned data items (Line 35) and reading the data items belonging to the snapshot determined by the begin timestamp (Line 36). When the transaction commits, the data site builds the new *depends* relationship following the *dependency recording rule* (Lines 38 to 42), and records this dependency information (Line 47). The data site then makes any updates visible by updating the partition version (Line 48). Finally, the adaptation advisor, updates the commit number (Line 56) followed by the session timestamp (Line 58) so that subsequent transactions abide by the commit order and avoid the long fork anomaly, and within a session clients avoid transaction inversions.

Appendix A.2 provides a proof of MorphoSys' concurrency control protocol provides SSSI in the presence of physical design changes.

4.3.3 Update Propagation

To support dynamic replication of lazily maintained partitions, MorphoSys propagates and applies updates on a per partition basis. If a transaction T updates a partition p then the data site maintains a redo buffer of T's changes. When T commits, for each updated partition p, the data site serializes the changes made to p, consisting of the updated data items, p's version number ($v(p)$), and the recorded dependencies (T^C). The data site writes this serialized update to a per partition *redo-log*. Data sites replicating p subscribe to the partition's log, asynchronously receive the serialized update, and apply the update as a *refresh transaction*.

Replicas ensure a snapshot consistent state by applying T's refresh transaction to partition p only after applying all previous updates to the partition, based on the partition version number. The refresh transaction installs T's updates to p by creating new versioned data items and makes the update visible by incrementing p's version number.

MorphoSys' design choices reduce the overhead of maintaining replicas because: (i) multi-versioning allows concurrent execution of read-only transactions and refresh transactions on replica partitions, and (ii) tracking transaction dependencies allows per partition execution of refresh transactions, which eliminates blocking on updates to other partitions.

4.3.4 Physical Design Change Execution

To execute the five physical design change operators efficiently, MorphoSys integrates them with the concurrency control algorithm and the update propagation protocol.

Adding and Removing Replicas

MorphoSys leverages multi-versioning to efficiently add a partition replica by snapshotting the partition's master state without acquiring locks. A data site snapshots a partition p by reading the last written position in its redo-log, the version number of the partition ($v(p)$), p's dependency information, and a snapshot of all data items in the partition at version $v(p)$. At the newly created replica, the data site installs this snapshot of the partition, records the version number, and subscribes to updates to the partition beginning from the last known position in the redo-log. At this point, the data site continues to apply propagated updates.

A data site removes a replica of a partition by stopping its subscription to the partition's updates. The data site then deletes the partition structure so future transactions do not access the partition. MorphoSys uses reference-counted data structures [88] to access partitions, which ensures that ongoing transactions can read from the removed partition replica without blocking the physical design change operation.

Splitting and Merging Partitions

MorphoSys' partition-based concurrency control executes the splitting and merging of partitions as transactions. These transactions update only partitions' metadata and not any data items. Hence, when splitting a partition p to create new partitions p_L and p_H, the data site mastering p acquires a partition lock on p. Thus, no other updates to p can take place during the split. Then, the data site creates new partitions p_L and p_H and assigns them both a version number equal to the version number of the original partition $v(p)$. The data site logically copies the reference-counted data items from p to the corresponding new partition, p_L or p_H. Committing the split operation induces a dependency between the three partitions (p, p_L, p_H) and results in p_L and p_H receiving p's dependency information, which ensures that transactional accesses to the new partitions observe a consistent state. On commit, the data site removes p. The bidirectional tracking of the *depends* relationship among partitions allows MorphoSys to split partitions without updating other partitions' states, thus reducing overhead.

Data sites replicating partition p observe the split operation in the redo-log. Replicas apply the split as a refresh transaction as outlined at the master data site, subscribe to updates of the newly created partitions p_L and p_H and remove their subscription to p. While a split is in progress, ongoing read transactions can access p but all subsequent transactions run on the newly created p_L and p_H.

Data sites merge partitions by following the reverse of splitting a partition, with two differences. First, when merging two partitions p_L and p_H to create partition p, the data site assigns the version of p as the maximum of p_L's and p_H's version numbers. Second, at a replica, the merge operation does not complete until both p_L and p_H are at the same state (as indicated by version numbers) as when the merge occurred at the master site.

Changing Partition Mastership

MorphoSys changes the mastership of a partition from site S_i to S_j as a metadata-only operation via update propagation of the mastership change information to all partition replicas, following the grant and release protocol described in Chapter 3.2.2. MorphoSys' concurrency control and update propagation protocol reduce blocking times when changing mastership as the new master waits only for the partition undergoing remastering to reach the correct state. By contrast, prior approaches [119, 12] require all data items in the system to reach the same, or later, state as the prior master.

4.4 Physical Design Strategies

As a workload executes, MorphoSys automatically decides: (i) how to alter its physical design with the aforementioned operators, and (ii) when to do so. The adaptation advisor makes these decisions by quantifying the expected benefit of the feasible design changes and greedily executing the set of changes with the greatest expected benefit. MorphoSys quantifies the benefit of a physical design change by modelling the effect that a design change will have on the future workload as well as the cost of performing the change. The adaptation advisor uses a workload model (Chapter 4.4.1) and learned costs of transactions and physical design changes (Chapter 4.4.2) to make its design decisions (Chapter 4.4.3).

4.4.1 Workload Model

MorphoSys continuously captures and models the transactional workload to make design decisions. The adaptation advisor samples submitted transactions and captures data item read and write access frequencies. MorphoSys maintains these statistics on a per partition basis, tracking per data item statistics only for frequently accessed partitions.

For a partition p, the adaptation advisor maintains the probability of reads $R(p)$, and writes $W(p)$, to the partition compared to all partition accesses. The adaptation advisor

Table 4.1: MorphoSys' cost functions and their use in predicting costs for transactions and physical design change operators.

Cost Function	**Arguments**	**Operations**					
		transaction	*remaster*	*split*	*merge*	*add_replica*	*remove_replica*
$\mathcal{F}_{service}(c_s)$	c_s = CPU utilization of site S	✓	✓	✓	✓	✓	✓
$\mathcal{F}_{wait\_updates}(v_n, v_r, u_s)$	v_n = partition version necessary v_r = version of partition at replica u_s = fraction of updates to partition compared to all replicas at site S	✓ (If reads)	✓				
$\mathcal{F}_{lock}(w)$	w = write contention of partitions	✓ (If writes)	✓	✓	✓		
$\mathcal{F}_{read\_write}(r_d, w_d)$	r_d (w_d) the number of data items read (written)	✓				✓	
$\mathcal{F}_{commit}(r_p, w_p)$	r_p (w_p) the number of partitions read (written)	✓	✓	✓	✓		

computes $R(p)$ (or $W(p)$) by dividing the per partition read (write) count by the running count of all reads and writes to all partitions.

As data item accesses are often correlated [12, 53, 172], MorphoSys tracks data item co-access likelihood. MorphoSys leverages these statistics to determine the effect of a proposed physical design change on future transactions (Chapter 4.4.3). Formally, $P(r(p_2)|r(p_1))$ represents the probability that a transaction reads partition p_2 given the transaction also reads p_1. I define similar statistics for write-write ($P(w(p_2)|w(p_1))$), read-write ($P(r(p_2)|w(p_1))$), and write-read ($P(w(p_2)|r(p_1))$) co-accesses.

MorphoSys adapts its model to changing workloads using adaptable damped reservoir sampling [29]. The reservoir determines sampling of transactions, generation of their access statistics and expiration of samples based on a configurable time window. MorphoSys uses statistics from these transactions in the reservoir to adapt its design to workload changes.

4.4.2 Learned Cost Model

Given a physical design, the adaptation advisor estimates the costs of executing transactions and applying physical design changes using a learned cost model. This cost model predicts the latency of design change and transaction execution operations. MorphoSys' implementation of these operations (Chapter 4.3) translates to a natural system latency decomposition: *waiting for service* at a site, *waiting for updates*, *acquiring locks*, *reading and writing* data items, and *commit*. Thus, MorphoSys decomposes the cost model into five corresponding cost functions (Table 4.1) that it learns and combines to predict the latency of operations.

The adaptation advisor learns these cost functions continuously using linear regression models because they are easy to interpret, do not require large amounts of training data, and are efficient for both inference and model updates [78]. In general, MorphoSys favour cost functions with few input parameters as such functions tend to be robust and accurate (Chapter 4.6.3). The adaptation advisor continuously updates the weights in the learned cost functions based on its predictions and corresponding observed latencies.

Next, I discuss the five cost functions in Table 4.1 and how MorphoSys combines them to predict latency of operations. Chapter 4.4.3 details how MorphoSys uses the cost functions to compute the expected benefit of different physical designs.

Waiting for Service

The load at a site determines how quickly the site services a request, with a site under heavy load taking longer. As MorphoSys is an in-memory system, and thus CPU bound [110, 77], it models data site load based on average CPU utilization, which the adaptation advisor polls for regularly. Hence, MorphoSys models the service time at a data site S as $\mathcal{F}_{service}(c_s = cpu\_util(S))$. To compute the service time, MorphoSys subtracts the latency of the operation measured at the data site from the observed latency at the adaptation advisor. Hence, network latency is captured in the service time. All operations take into account the service time at a site. In the case of operations that involve multiple sites (*remaster*, *add_replica*), MorphoSys computes $\mathcal{F}_{service}$ for each involved site.

Waiting for Updates

Recall that remastering does not complete until all necessary updates have been applied to the partition being remastered (Chapter 4.3.4). Similarly, transactions may wait when reading a replica partition to observe a consistent state. The waiting time depends on the number of propagated updates needed and the relative frequency of those updates, both of which implicitly include an estimate of the network latency. MorphoSys models the waiting time at a partition p as $\mathcal{F}_{wait\_updates}(v_n, v_r, u_s)$, where v_n is the version of the partition required at the replica, v_r is the current version of the replica partition, and u_s is the fraction of updates applied at the replica S relevant to partition p. MorphoSys determines u_s using Equation 4.2; the denominator represents the total likelihood of updates to partitions that have replicas at S, which MorphoSys aggregates.

$$u_s = \frac{W(p)}{\sum_{p_i : S \in replicas(p)} W(p_i)} \tag{4.2}$$

MorphoSys uses the session timestamp to determine v_n for transactions while remastering uses the latest polled version from the old master. As data sites apply updates in parallel that a transaction may require, MorphoSys considers the largest predicted $\mathcal{F}_{wait\_updates}$ from all partitions in the read set.

Acquiring Locks

Recall from Chapter 4.3.2 that update operations acquire per partition locks. Hence, increased write contention on a partition results in longer partition lock acquisition time. Thus, MorphoSys predicts lock acquisition time as $\mathcal{F}_{lock}(w)$ where w represents write contention. MorphoSys estimates w using $W(p)$, the probability of writes to a partition. For operations that access multiple partitions such as transactions or partition merging, data sites acquire locks sequentially. MorphoSys defines $w = \sum_p W(p)$ for all relevant p, i.e., partitions p_L, p_H for *merge*, and all partitions in a transaction's write set.

Reading and Writing Data Items

The primary function of transactions is to read/write data. Hence, MorphoSys estimates the time spent reading and writing data items using the cost function $\mathcal{F}_{read\_write}(r_d, w_d)$, where r_d and w_d represent the number of data items read and written, respectively. The adaptation advisor determines r_d and w_d from a transaction's read and write set. Adding a replica of a partition requires reading a snapshot of the partition and installing it into the newly created replica. Hence when estimating the latency of adding a replica, MorphoSys also considers $\mathcal{F}_{read\_write}$ using the number of data items in the partition as the number of data items read and written.

Commit

Data sites commit operations by updating partition version numbers and dependency information (Chapter 4.3). The number of updates depends on the number of partitions involved in the operation. MorphoSys uses the number of partitions read (r_p) and written (w_p) by the operation to estimate the commit time as $\mathcal{F}_{commit}(r_p, w_p)$. Given that adding and removing a replica of a partition does not entail committing, MorphoSys omits commit latency for these operations.

Putting it Together

The adaptation advisor predicts the latency of each operation by summing the values of the cost functions shown in Table 4.1 that are associated with the operation. For example, to predict the latency of a *split* at a site, MorphoSys uses the recorded CPU utilization of the site (c_s) and write contention of the partition (w) to compute $\mathcal{F}_{service}(c_s) + \mathcal{F}_{lock}(w) + \mathcal{F}_{commit}(0, 3)$. The system tracks the latency of the entire split operation, as well as the portion of the time spent in the network waiting for service, locking and committing. Given these observed latencies and latency predictions (e.g. $\mathcal{F}_{service}(c_s)$), the adaptation advisor uses stochastic gradient descent [167] to update its cost-functions.

4.4.3 Physical Design Change Decisions

MorphoSys makes physical design and routing decisions to improve system performance and to execute transactions at one site. The adaptation advisor considers each data site as a candidate for transaction execution and develops a physical design change *plan* for the site. A site's plan consists of a set of physical design change operators that allow the transaction to execute at the site, including adding or removing replicas, splitting or merging partitions, and remastering. For a data site S, the adaptation advisor computes the cost of executing the plan, $C(S)$, and expected benefit, $E(S)$. The adaptation advisor selects the data site, and plan, that maximize the net benefit $net(S) = \lambda \cdot E(S) - C(S)$. The magnitude of $\lambda(> 0)$ controls the relative importance of the expected benefit of the plan. I will use the *split* operation of partition $(0, 9)$ from Figure 4.1b as a running example to illustrate MorphoSys' decisions.

The cost of executing a plan $C(S)$ is the sum of the execution costs of each of the plan's operators, as defined in Chapter 4.4.2. A low execution cost indicates that the plan will execute quickly. The input parameters for these costs correspond to the statistics from the existing physical design. In the *split* operator example, MorphoSys estimates $C(S)$ by computing $\mathcal{F}_{service}(c_1) + \mathcal{F}_{lock}(W_{(0,9)}) + \mathcal{F}_{commit}(0, 3)$, where c_1 represents the CPU utilization at data site 1 and $W_{(0,9)}$ represents contention of the partition being split.

To determine the expected benefit $E(S)$, MorphoSys predicts the latency of transactions under the current physical design and subtracts the predicted latency of transactions under the physical design that would result from executing the plan. A good plan decreases predicted latencies, which increases $E(S)$ and thus increases $net(S)$. To determine $E(S)$, MorphoSys samples transactions and uses the learned cost model with input parameters that correspond to the plan's new physical design of the database.

Consider the effect splitting partition $(0, 9)$ has on transactions T_1 and T_2 from Figure 4.1b. First, MorphoSys predicts the latency of T_1 and T_2 executing at site 1 under the current physical design. As both transactions update partition $(0, 9)$, MorphoSys estimates $\mathcal{F}_{lock}(W_{(0,9)})$ as part of the transaction latency, which I call $L_{current}$. MorphoSys then estimates the effect of splitting the partition into $(0, 5)$ and $(6, 10)$, each of which has less write contention than partition $(0, 9)$. Then, MorphoSys uses the contention of partitions $(0, 5)$ and $(6, 10)$ to estimate the latency of transactions T_1 and T_2 on the future design, which I call L_{future}. Finally, MorphoSys computes $E(S)$ as $L_{current} - L_{future}$. If $E(S)$ is positive, then the design change is predicted to reduce latency and improve system performance. Design changes with non-positive $E(S)$ values are unlikely to improve system performance and are thus avoided.

Sampling Transactions

Computing the expected benefit of a design change plan ideally requires knowledge of future transactions to be submitted to the system, which is not available. Thus, MorphoSys draws samples of transactions from its reservoir of previously submitted transactions to emulate a workload of future transactions, for example T_1 and T_2, from the running example. MorphoSys also generates emulated transactions based on its workload model to ensure robustness in design decisions. To generate these transactions, MorphoSys selects a partition p_1 at random following the partition access frequency distribution. Then, MorphoSys samples a second partition p_2 co-accessed with p_1 and generates four transactions that access p_1 and p_2, based on all combinations of read and write co-accesses, and weight any expected benefit by the likelihood of co-access.[1] If MorphoSys generates a transaction T that reads p_1 and updates p_2, then it weighs the expected benefit to T by $R(p_1) \times P(w(p_2)|r(p_1))$, when computing $E(S)$. MorphoSys selects data item accesses to the partition uniformly at random.

Adjusting Cost Model Inputs

The adaptation advisor predicts the latency effect of physical design changes by predicting changes to inputs of its cost model. This is done by considering how design changes affect CPU utilization, update application rate, and contention.

Recall that $\mathcal{F}_{service}(c)$ predicts the time spent waiting for a data site to service a request. Data sites use resources to perform database reads, writes, and apply propagated updates.

[1] p_1 and p_2 can represent the same partition if a transaction accesses the same partition more than once.

MorphoSys predicts CPU utilization based on the frequency of reads (r_S), writes (w_S), and propagated updates applied (u_S) using the cost function $\mathcal{F}_{CPU}(r_S, w_S, u_S)$. Remastering a partition p, and adding or removing partition replicas affect r_S, w_S and u_S based on the probability of reads and writes to p. Morphosys uses the output of $\mathcal{F}_{CPU}$ as input to $\mathcal{F}_{service}$ when predicting the time spent waiting for a data site to service a transaction. By considering $\mathcal{F}_{service}$, MorphoSys favors designs that distribute the load to all data sites, which minimizes wait time.

Splitting a partition reduces the probability of reads and writes to the newly created partitions, which store fewer data items. The reverse holds for merging partitions. To reduce tracking overheads, MorphoSys assumes uniform accesses to data items within a partition when modelling the effects of a design change on write contention (w in $\mathcal{F}_{lock}$), and update frequency (f_S in $\mathcal{F}_{wait\_updates}$). If the design change occurs, then over time, the partition statistics reflect the partition access likelihood. Partition splits and merges also change the number of partitions accessed by a transaction, which MorphoSys considers when predicting commit latency ($\mathcal{F}_{commit}$).

A physical design change may enable future transactions to execute at a single site without further design changes. MorphoSys encourages such design changes as they account for data locality in the workload by incorporating the expected benefit of not needing future design changes. To do so, MorphoSys predicts the latency of previously required physical design changes, which the plan saves, and adds these savings to the expected benefit. Conversely, MorphoSys discourage plans that induce designs precluding single-site transactions by subtracting the predicted latency of future physical design changes from the plan's expected benefit. Hence, MorphoSys avoids generating design changes that it could shortly undo.

Generating Plans

The adaptation advisor generates a physical design change plan for a site by adding operations necessary for single-site transactions: remastering and adding replicas of partitions in the write and read sets, or removing replica partitions to satisfy space constraints. The adaptation advisor then adds further beneficial design changes to the plan: splitting or merging partitions, and remastering or adding replicas of partitions co-accessed with written and read partitions. The partition split in Figure 4.1b is an example of a beneficial design change, while the partition remaster in Figure 4.1d is necessary for single-site execution.

The adaptation advisor considers partitions in the transaction's read or write set as candidates for splitting or merging. If a partition in the read or write set has above-average access probability, indicating contention on the partition, and the split is beneficial then the adaptation advisor adds the partition split to the plan. Conversely, merging infrequently accessed partitions reduces the number of partitions in the system, which reduces metadata overheads. Thus, if a partition in the read or write set, and one of its neighbouring partitions, have below-average access likelihood and it is possible and beneficial to merge the two partitions, then MorphoSys adds the merge operation to the plan. Considering access frequencies ensures that MorphoSys does not undo the effects of a split with a merge, or vice versa, in the immediate future unless the access pattern changes significantly.

When a plan for a site includes remastering, addition or removal of a partitions replica, MorphoSys *piggybacks* other design change operations for correlated partitions. Piggybacking operations promotes data locality and future single-site transactions. For example, when remastering partition p_1, MorphoSys tries to piggyback the remastering of a partition p_2 frequently co-written with p_1, which occurs when both of $P(w(p_1)|w(p_2))$ and $P(w(p_2)|w(p_1))$ are high. This probability-driven remastering with piggybacking amortizes partition remastering cost while promoting co-location of co-accessed partitions. Similarly, if remastering p_1, MorphoSys will try to piggyback replica addition of a frequently read partition p_2 if $P(w(p_1)|r(p_2))$ and $P(r(p_2)|w(p_1))$ are high.

MorphoSys removes replica partitions when they are no longer beneficial to maintain or to satisfy memory constraints. If a data site is within 5% of its memory limit, then the adaptation advisor removes replica partitions by computing the expected benefit of removing a partition and iteratively removes partitions with the least expected benefit first until the memory constraint is satisfied.

MorphoSys' generated plans use the cost and workload models to produce design changes that reduce contention, encourage data locality and single-site transactions, and maximize expected benefit when compared to execution costs — all without immediately undoing or redoing changes.

4.5 The MorphoSys System

I implemented MorphoSys following the architecture described in Chapter 2. The implementation includes (from Chapters 4.3 and 4.4) the concurrency control protocol, update propagation scheme, physical design change operators, workload model, learned cost model, and physical design strategies.

Following the architecture from Chapter 2, MorphoSys uses Apache Kafka [102] as the redo-log, and cooperatively applies propagated updates at data sites. Application of updates is shared between the thread receiving updates from the Kafka redo-log and transaction execution threads waiting for specific partition versions.

The adaptation advisor tracks the state of each data partition in per table concurrent hash-table structures for efficient lookups. This state contains a partitions range of data items, the location of its master copy, its replica locations, and partition access frequencies. The adaptation advisor updates partition state upon the successful completion of a design change. To minimize latency, the adaptation advisor executes design change plan operations in parallel.

4.6 Experimental Evaluation

I now present experimental results that show that MorphoSys' ability to dynamically change the physical design of a database significantly improves system performance.

4.6.1 Evaluated Systems

I evaluated MorphoSys against five alternative distributed database systems that employ state-of-the-art dynamic, or popular static, physical designs. I implemented these comparative systems in MorphoSys using strong-session snapshot isolation and multi-version concurrency control to ensure an apples-to-apples comparison. Recall that MorphoSys is designed to deliver superior performance irrespective of its initial physical design. Thus, in each experiment, MorphoSys starts with an initial physical design containing *no replicas* and a *randomized* master placement of partitions, an *unknown* workload model, and *must learn* its cost model from scratch. By contrast, as described next, I advantaged MorphoSys' competitors by using *a priori* knowledge of the workload to optimize their initial physical design.

Clay dynamically partitions data based on access frequency to balance load [172]. Clay performs this repartitioning periodically, with a default period of 10 seconds. Unlike MorphoSys, Clay does not replicate data and uses 2PC to execute transactions that access data mastered at multiple sites [172]. Clay begins with an initial master placement so that each site masters the same number of data items, such as by warehouse in TPC-C [172].

Adaptive Replication (ADR) is a widely used algorithm to dynamically determine what data to replicate, and where [201, 82, 131, 139]. Like Clay, ADR uses 2PC for multi-site transactions. I advantaged ADR with offline master partition placement using Schism

[53]. Schism is a state-of-the-art offline tool that uses a workload's data access patterns to generate partitioning and placement of master copies of data items and their replicas such that distributed transactions are minimized while distributing load.

Single-Master and **Multi-Master** as described in Chapter 3.5.1.

DynaMast, presented in Chapter 3, is a fully-replicated database that dynamically remasters data items to guarantee single-site transactions [12]. Unlike MorphoSys, DynaMast does not dynamically replicate or partition data. DynaMast begins each experiment with the same starting master placement as Clay in which masters are uniformly placed among sites.

Finally, I compared MorphoSys with the off-the-shelf commercial version of the partitioned, database system **VoltDB** that uses a static physical design [187]. I favoured VoltDB using Schism's initial design and followed VoltDB's benchmarking configuration [165] that reduces durability to optimize performance by disabling synchronous commit to reduce commit latency and delaying snapshots to reduce overheads. VoltDB is also configured to use the maximum available hardware resources on each node as well as the optimized executable.

4.6.2 Benchmark Workloads

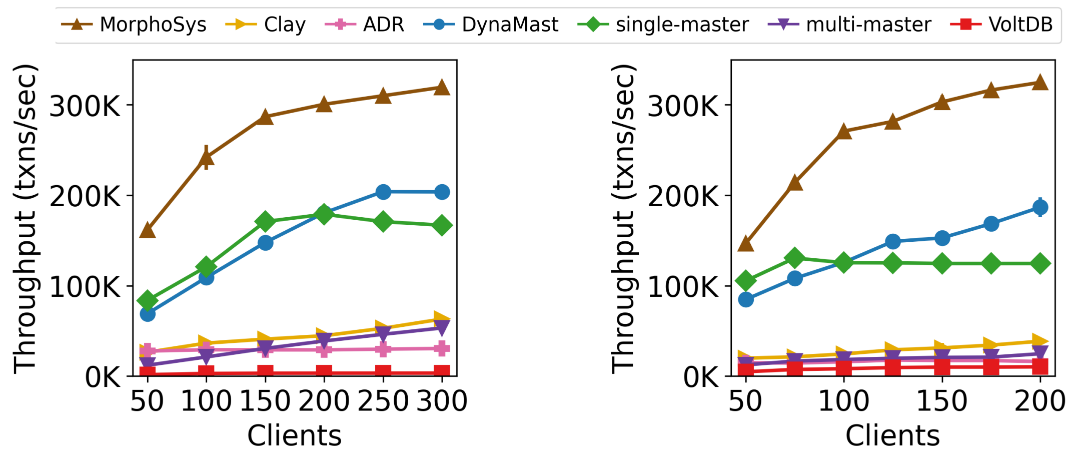
The experiments execute on up to 16 data sites with the same configuration as described in Chapter 3.5.2.

I conducted experiments using the YCSB, TPC-C, Twitter and SmallBank benchmark workloads [58, 48, 1] as they contain multi-data item transactions and access correlations representative of real-world workloads. YCSB, TPC-C and Smallbank are configured as described in Chapter 3.5.2.

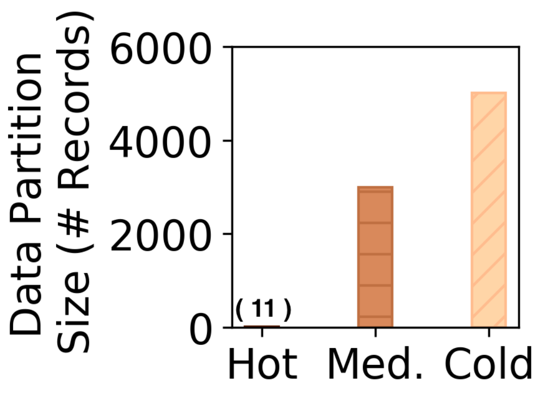
Twitter models a social networking application, featuring heavily skewed many-to-many relationships among users, their tweets, and followers. Consequently, Twitter's predominantly read-intensive workload (89%) contains transactions with complex accesses spread across the social graph data.

4.6.3 Results

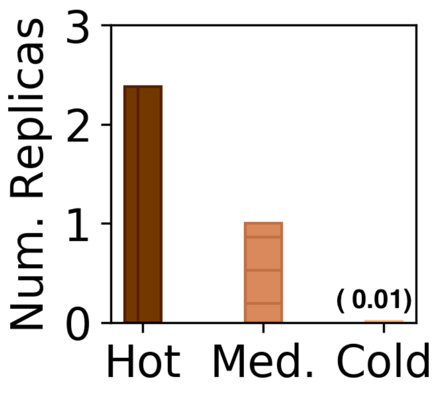
I now discuss MorphoSys' experimental results for varying access patterns, load, and read-write mixes.



(a) Read-Mostly (Skew) (b) Write-Heavy (Skew)



(c) Partition Size



(d) Replication Factor

Figure 4.2: MorphoSys Performance results for Skewed YCSB.

Workloads With Skew

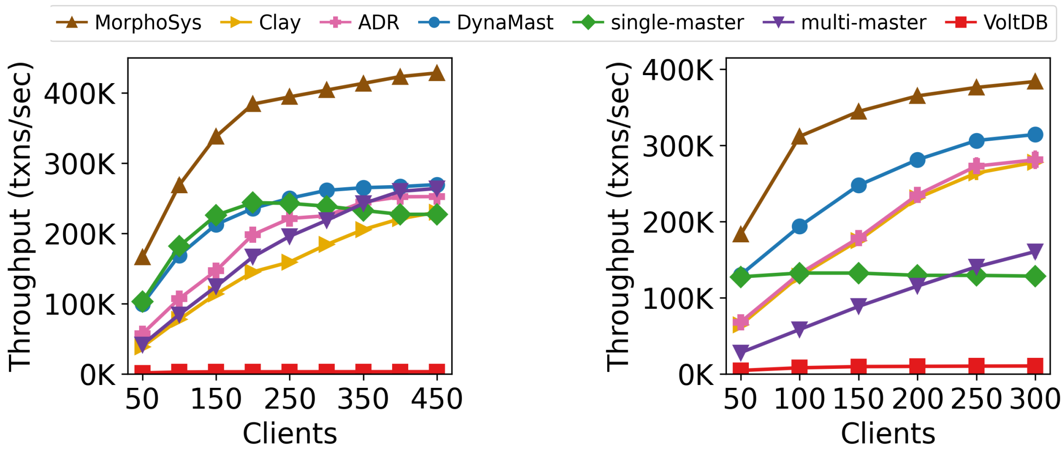
Skewed workloads generate contention and load imbalance, both of which MorphoSys mitigates using dynamic physical design changes. To evaluate MorphoSys' effectiveness under skewed data accesses, I used YCSB workloads with read-mostly (50% scans, 50% multi-key RMW) and write-heavy (10% scans, 90% multi-key RMW) transaction mixes with Zipfian access skew. Throughput results for both workloads (Figures 4.2a and 4.2b) demonstrate that MorphoSys delivers significantly better performance, about 98× to 1.75× higher throughput than the other systems as it dissipates contention and balances load by dynamically repartitioning heavily accessed and contended data items into smaller partitions.

In Figure 4.2c, I classify data items as hot if they are in the top 10% of the most frequently accessed data, medium if in the next 30%, and the remaining as cold data. As Figure 4.2c shows, on average, MorphoSys groups the hottest data items into small partitions containing up to 100 data items. By contrast, MorphoSys groups infrequently accessed or cold data items into partitions that, on average, contain 5000 records. Such dynamic partitioning of data reduces lock acquisition time by more than a factor of 7, from nearly 850 μs to 110 μs compared to the other systems.

In addition to dynamic partitioning, MorphoSys employs dynamic replication. Figure 4.2d shows the replication factor for data items with different access frequencies. In contrast to the fully replicated DynaMast, single-master and multi-master architectures, MorphoSys replicates frequently accessed data items but avoids replicating infrequently accessed data. By keeping, and maintaining, fewer replicated data items, MorphoSys uses less compute resources to apply propagated updates, thereby freeing up resources to improve throughput by 2.6$\times$ over the single-master architecture in the write-heavy workload (Figure 4.2b). Although multi-master fully replicates data, and ADR dynamically adds replicas of frequently read partitions, they both suffer heavily from the compounding effects of contention as a result of static partitioning and distributed transaction coordination. MorphoSys combines both dynamic replication and partitioning while guaranteeing single-site transactions resulting in throughput improvements of up to 13$\times$ over ADR and multi-master.

Both Clay and MorphoSys dynamically partition to mitigate the effects of contention. However, unlike Clay, MorphoSys replicates hot partitions frequently to distribute read load among sites and make remastering more efficient. MorphoSys groups together colder co-accessed data items to reduce the metadata needed to track partitions, dependencies, and version histories. MorphoSys converges to its final design faster than Clay as MorphoSys uses every transaction as an opportunity to make design changes while guaranteeing single-site execution, in contrast to Clay's periodic operation and use of expensive 2PC. Thus, MorphoSys improves throughput over Clay by 8.5$\times$ and 5$\times$ for the update-intensive (Figure 4.2b) and read-heavy (Figure 4.2a) workloads, respectively. The scan-heavy workload exacerbates the effects of distributed reads in VoltDB as it requires enqueueing the scan operator on every site blocking all other transactions from executing due to VoltDB's single-threaded execution model [165, 187]. VoltDB's static design cannot adapt to heavy skew effects, unlike MorphoSys that improves throughput over VoltDB by almost 100$\times$.

By taking a holistic approach to dynamic physical design and considering all 3 factors, namely, partitioning, replication and mastering, MorphoSys outperforms its competitors



(a) Read-Mostly (Uniform) (b) Write-Heavy (Uniform)

Figure 4.3: MorphoSys Performance results for Uniform YCSB.

that consider only one of these multiple aspects of design.

Workloads with Uniform Access Patterns

Next, I evaluate MorphoSys in the presence of a uniform access pattern. In this workload, MorphoSys groups data items into partitions containing approximately 3000 data items, and on average replicates these partitions to one replica. This partial replication reduces the computational resources needed to maintain replicas when compared to fully replicated systems (single-master, multi-master and DynaMast). Hence, in the read-mostly case (Figure 4.3a), MorphoSys improves throughput over fully replicated systems in the range 1.85× to 1.6×.

A replica partition in MorphoSys supports read transactions and flexibly provides mastership opportunity when deciding on master placement. MorphoSys uses this flexibility to guarantee single-site transactions and to judiciously amortize the cost of design changes. As such, MorphoSys eliminates costly distributed coordination that Clay, ADR, VoltDB, and multi-master continually incur.

Clay initiates data repartitioning only when it detects an imbalance in partition accesses. In this workload, Clay rarely detects any imbalance in accesses, and thus rarely repartitions data. Hence, Clay must continually execute costly multi-site transactions, which in the case of scans, are susceptible to stragglers. By contrast, MorphoSys performs physical design changes as transactions execute, and co-locates co-accessed data together via dynamic replication and mastering to execute single-site transactions.

ADR dynamically adds replicas, which allows for efficient single-site scan execution, thus improving performance over Clay but falls short of MorphoSys. By considering master placement, replication, and data partitioning, MorphoSys improves throughput over ADR and Clay by almost 1.9×.

Next, I stress the systems' ability to balance update load among data sites using a write-heavy YCSB workload (Figure 4.3b). MorphoSys achieves even load distribution among sites in the distributed system by predicting the time spent waiting for service at a site, which is primarily determined by site load (Chapter 4.4.2). Evenly distributing load improves throughput by 3× as compared to single-master, which executes all updates at one master site. By contrast, multi-master must rely on Schism's offline analysis to ensure even routing of requests among all sites. Despite balancing the load, multi-master fully replicates data and requires distributed transaction coordination; thus, MorphoSys improves throughput over multi-master by 2.4×. ADR does not frequently replicate in this write-heavy workload and improves performance compared to multi-master, but MorphoSys remains unmatched outperforming ADR by 1.35×. The shorter read-modify-write transactions improve VoltDB's throughput compared to the scan-heavy workload. However, like ADR, VoltDB must coordinate transactions with 2PC, hence MorphoSys improves throughput over VoltDB by over 38×.

Clay and DynaMast both aim to balance update load, but come with significant shortcomings that MorphoSys addresses. As in the read-mostly case, MorphoSys' dynamic and partial replication of data reduces the overhead of maintaining replicas that DynaMast incurs. While doing so, MorphoSys still ensures the flexibility necessary to support dynamic mastership placement. Clay makes a static replication decision of never to replicate, but suffers as it does not reduce distributed transaction coordination through effective master placement. In this update-intensive uniform workload, MorphoSys' comprehensive physical design strategies and efficient execution result in 1.4× and 1.2× throughput improvement over Clay and DynaMast, respectively.

Workloads with Complex Transactions

I now focus the evaluation on TPC-C, a workload that contains complex transactions simulating an order-entry application. This workload features correlated data accesses to warehouses and districts. However, MorphoSys has no knowledge of this pattern and must learn this workload model and create an effective distributed physical design. Fig-

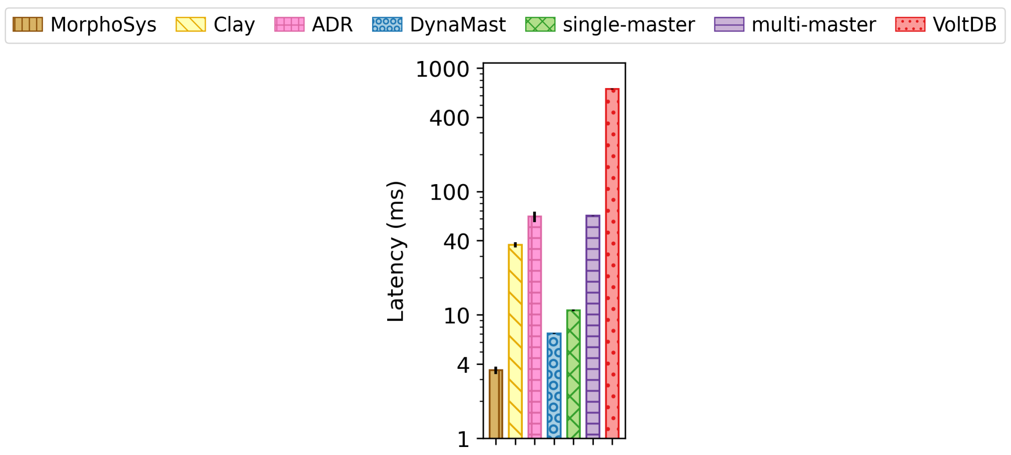
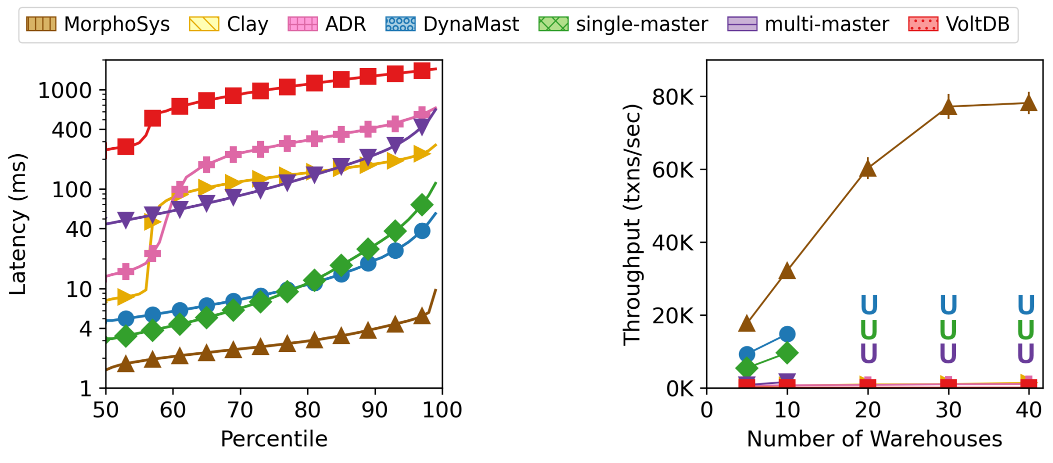


Figure 4.4: Average Latency in the TPC-C workload.



(a) TPC-C Tail Latency

(b) TPC-C Throughput

Figure 4.5: MorphoSys' Performance results for the TPC-C workloads. In Figure 4.5b, U's indicate that fully replicated systems (DynaMast, single-master and multi-master) were *unable* to run due to memory constraints.

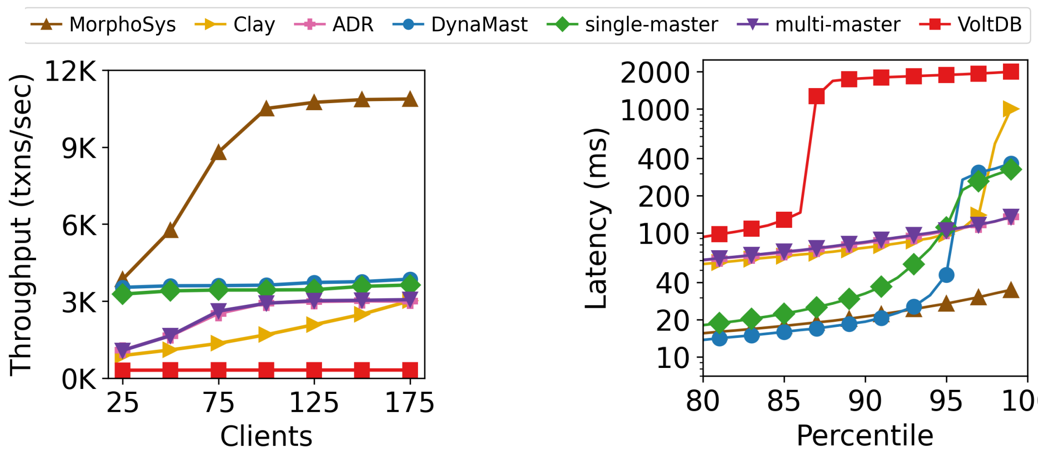
ure 4.4 shows that MorphoSys has the lowest average latency of transactions in the TPC-C workload, reducing latency by between 99% and 50% over its competitors. MorphoSys significantly reduces tail latency, as shown in Figure 4.5a, with reductions ranging from 167× to 6×.

To understand why MorphoSys reduces latency, I examined the physical design decisions made by MorphoSys as well as the core properties of the *New-Order* transaction that has the highest latency of all TPC-C transactions. The *New-Order* transaction reads the warehouse data item to determine the purchase tax and updates the district with the next order identifier. Schism determines that the best master placement is based on the warehouse identifier of the district and customer as 90% of *New-Order* transactions access data local to the customer's warehouse. The remaining 10% execute *cross-warehouse* transactions.

Examining the physical design decisions made by MorphoSys reveals that it creates single data item partitions for the warehouse and district tables, and replicates these partitions to multiple sites. This data partitioning supports parallel execution of *New-Order* transactions on different districts, while replication ensures efficient dynamic mastering for cross-warehouse transactions. Additionally, I observed that MorphoSys heavily replicates the read-only items table as its cost model correctly predicts that there is little overhead required to maintain this table. MorphoSys selectively replicates the remaining tables including the frequently updated customer and stock tables. For these tables, MorphoSys adds and removes replicas of partitions to balance system load and ensure single-site transaction execution.

The TPC-C results show the benefit of MorphoSys' comprehensive physical design strategies. MorphoSys' dynamic formation of data partitions produces per district partitions that reduces contention when assigning the next order identifier and decreases *New-Order* transaction latency. Increasing the scale factor which controls the number of warehouses while maintaining constant contention allows for increased load, which improves throughput. However, increasing the scale factor increases the amount of data stored in the system. Partial and dynamic replication allow MorphoSys to selectively replicate and store more data than fully replicated systems. Thus, MorphoSys supports a higher scale factor and improves throughput over the fully replicated systems by between 48× to 5.5× (Figure 4.5b). By considering master placement and guaranteeing single-site transactions, MorphoSys improves throughput over systems that require distributed transactions by 900× to 48×.

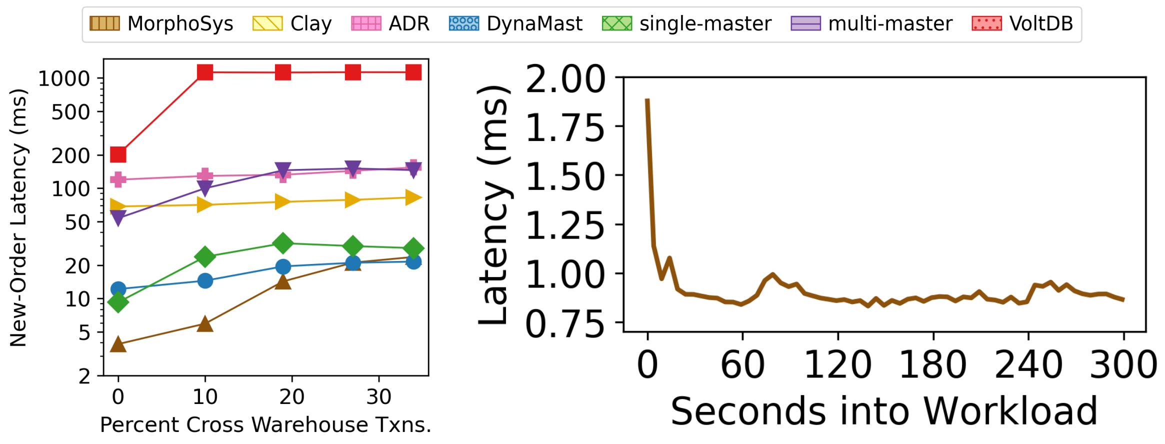
Figures 4.6a and 4.6b show the throughput and tail latency for the Twitter workload. MorphoSys achieves between 32× and 3× greater throughput than its competitors and reduces tail latencies by between 58× and 4×. MorphoSys' tail latency reductions primarily come from reducing the time spent waiting for updates to tweets in the *GetTweetsFromFollowing* transaction by maintaining per partition update queues. As in the read-mostly YCSB workload, ADR and multi-master behave similarly, with ADR replicating the most frequently accessed data. However, MorphoSys guarantees single-site reads, unlike ADR, and hence does not suffer from straggler effects due to multi-site scans. As Clay and



(a) Throughput

(b) Tail Latency

Figure 4.6: Performance results for the Twitter workload.



(a) New-Order Latency

(b) Adaptivity Over Time

Figure 4.7: MorphoSys' Adaptivity in Changing Workloads

VoltDB cannot replicate, they suffer even more from these straggler effects.

Adaptivity

Next, we present MorphoSys' ability to adapt to workload change in terms of shifting data accesses and load imbalance. Such workload shifts occur, for instance, due to changes in trends on a social network, or shifts in popularity of stocks on the stock exchange [203, 109, 86, 64, 159, 154].

The TPC-C experiment in Figure 4.7a shows how MorphoSys' adaptivity allows it to cater to different workloads. The figure depicts *New-Order* latency as we increase the percentage of cross-warehouse transactions. When the percentage is zero, MorphoSys' latency is one-quarter of its closest competitor. When the percentage of cross-warehouse transactions reaches about one-third, data locality decreases and MorphoSys' latency approaches that of single-master's, decreasing MorphoSys' relative benefit. Thus, when data locality is low, MorphoSys adapts its physical design by increasingly mastering data at a single site to avoid undoing and redoing physical design changes while distributing load among sites as much as possible.

To highlight MorphoSys' ability to react to workload changes, we experimented with a shifting hotspot [191], a phenomenon that frequently occurs in transactional workloads [85, 137]. We induce hotspots with a skewed YCSB workload and shift the center of the skew to a different part of the database every 60 seconds. This challenging workload causes MorphoSys to change its physical design to mitigate the effects of skew and load imbalance. Figure 4.7b shows the average latency over five minutes and four workload shifts. MorphoSys learns its initial design within the first 30 seconds, at which time MorphoSys reaches its minimum latency that is a reduction of the initial latency by almost 60%, illustrating the benefit of design changes. When a hotspot shifts, MorphoSys' latency increases by at most 20% from its minimum, returning to the minimum latency within 20 seconds on average as MorphoSys quickly splits partitions, adds replicas of the hot partitions and remasters to balance the load and mitigate the hotspot. These results show MorphoSys' effectiveness to rapidly adapt to workload changes due to its learned workload and cost models, and low overhead design changes.

Scalability

To measure MorphoSys' scalability, I scale the number of data sites from 4 to 16 in increments of 4 while also scaling the number of clients (60 per data site), and measure peak throughput using the read-heavy uniform YCSB workload. As shown in Figure 4.8, MorphoSys improves throughput by nearly 3× as the number of data sites grows from 4 to 16. MorphoSys achieves this near-linear scalability because its dynamic physical design effectively distributes transaction load among sites, minimizes replication overhead, eliminates distributed transaction coordination, and, as I show next, has low overhead.

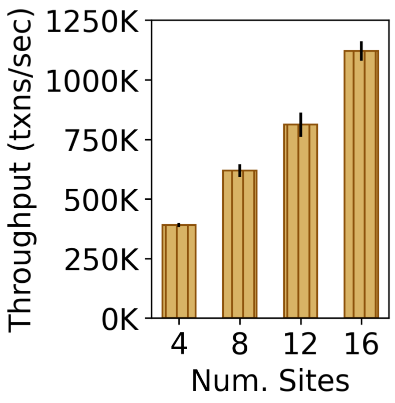


Figure 4.8: MorphoSys' Scalability using YCSB.

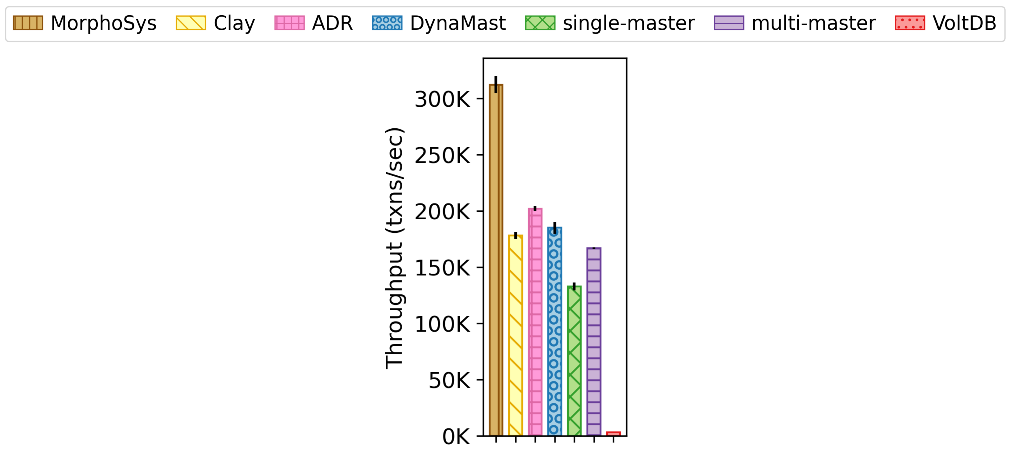


Figure 4.9: SmallBank Throughput in MorphoSys

Overhead and Model Accuracy

To understand MorphoSys' overheads, including planning and executing physical design changes, I evaluated performance using the SmallBank benchmark. Transactions in SmallBank access at most two data items using a uniform data access pattern. Thus, the time spent executing transaction logic is small, making it easier through relative comparison to identify where time goes in the system. Figure 4.9 shows the maximum throughput for the SmallBank workload. Observe that MorphoSys outperforms its competitors by between 104× and 1.5×, indicating that MorphoSys' dynamism incurs little overhead.

The performance of VoltDB is severely limited by distributed update transactions and the single-threaded execution model that blocks non-conflicting transactions belonging to the same thread of execution.

Table 4.2: Transaction latency breakdown within MorphoSys.

Operation	**Avg. Latency**	**Percent of Txn. Time**
Locating Partitions	27 ± 1.0 μs	2.1%
Plan Generation	49 ± 9.2 μs	3.8%
Workload & Cost Model	13 ± 6.8 μs	1.0 %
Design Change	42 ± 2.9 μs	3.3 %
Network & Queuing	252 ± 51 μs	19.8%
Locking	114 ± 39 μs	8.9%
Waiting for Updates	65 ± 5.9 μs	5.1%
Transaction Logic	550 ± 97 μs	43.0%
Commiting	158 ± 8.4 μs	12.4%
Total	1270 ± 310 μs	100%

Table 4.2 breaks down SmallBank transaction latency in MorphoSys. Observe that the system spends the plurality of time (43%) executing transaction logic. At data sites, just 25% of overall transaction latency is spent on MorphoSys' concurrency control, including waiting for any necessary updates, locking, and recording dependencies during commit. Given the small transaction footprint, this translates to low overhead as the latency is comparable to the amount of time that transactions spend in the network. This low latency is a consequence of the partition-based concurrency control and update propagation scheme. Finally, transactions spent just 10% of their time at the adaptation advisor, including an average of just 3.3% of time executing physical design changes. This small overhead results from MorphoSys amortizing the cost of design changes over many transactions, and executing design changes in parallel when they do occur.

Table 4.3 shows the relative frequency and average latency of each of the physical design operators. On average, MorphoSys executes a physical design change operator 30 times for every 1000 transactions, taking just over 6 ms to execute. The most expensive physical design change operators require physical copying of data, as in the case of adding a replica of a partition or waiting for all updates to arrive at the soon to be designated new master.

Recall from Chapter 4.4.2 that MorphoSys uses a cost model to predict operation latencies. Figure 4.10 examines the cost model's accuracy by comparing the actual and

Table 4.3: Design change operator frequency and latency in MorphoSys.

Operator	**Frequency** (per 1000 Txns.)	**Avg. Latency**
split	3.9 ± 0.3	3.7 ± 0.67 ms
merge	0.15 ± 0.06	8.3 ± 1.7ms
remaster	14.4 ± 0.5	33.3 ± 2.3 ms
add_replica	12.1 ± 0.4	40.4 ± 0.4 ms
remove_replica	0.12 ± 0.01	0.79 ± 0.04 ms
Total	30.7 ± 1.2	6.1 ± 11.7 ms

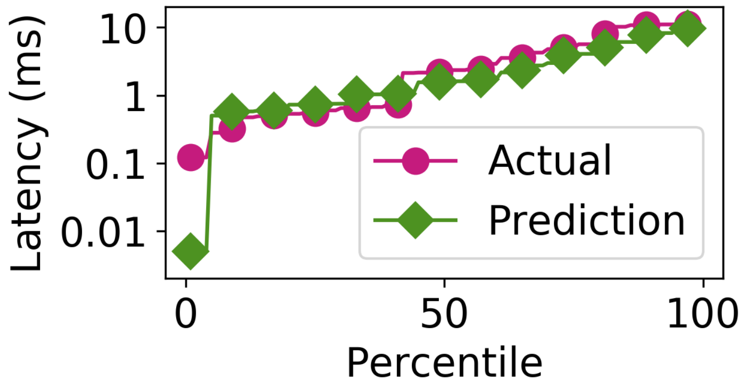


Figure 4.10: MorphoSys' Prediction Accuracy

predicted execution costs (latencies) of the *split* design change operator. The predicted latency closely tracks the actual latency with a coefficient of determination (R^2) of 0.81. This result indicates that the cost model captures design change costs with high accuracy while being easy to interpret and efficient to train.

Different Intial Physical Designs

In the experiments in Chapter 4.6 I provided ADR, VoltDB, and multi-master with offline *a priori* knowledge of the workload, when initializing the physical design, by using Schism [53] (Chapter 4.6.1). I advantaged DynaMast and Clay by balancing the number of partitions mastered at each site. However, for MorphoSys, the initial physical design was completely random. To examine the effect of these different initial physical designs on performance, I ran experiments with the skewed read-mostly YCSB workload and mea-

Table 4.4: YCSB Read-Mostly Skew Tput, under different initial physical designs: Schism [53] and random. We additionally show the time the system takes to converge to peak throughput for each initial physical design. Single-master uses the same physical design for both Schism and random, as it always places master copies of partitions on a single node.

System	**Schism Avg. Tput (txn/sec)**	**Random Avg. Tput (txn/sec)**	**Schism Time to Peak Tput**	**Random Time to Peak Tput**
MorphoSys	**309 ± 11k**	**302 ± 11k**	6 s	23 s
Clay	61.7 ± 2.9k	41.6 ± 1.5k	3 s	58 s
ADR	29.7 ± 3.0k	18.5 ± 1.3k	3 s	82 s
DynaMast	210 ± 3.5k	198 ± 2.2k	5 s	122 s
single-master	170 ± 1.5k	168 ± 3.5k	2 s	2 s
multi-master	46.1 ± 3.4k	21.8 ± 1.1k	4 s	9 s
VoltDB	3.21 ± 0.02k	1.98 ± 0.02k	7 s	6 s

sured peak throughput under two initial physical designs: Schism and a randomized design. Additionally, for the initial randomized design, I measured the time it took for the system to converge to within the confidence interval of its peak throughput. These results are presented in Table 4.4.

As shown in Table 4.4, MorphoSys has the highest throughput under both initial physical designs. Furthermore, with a randomized initial physical design, MorphoSys converges quickly (in 23 seconds) to within 2% of its throughput when initialized with *a priori* knowledge. Without a priori workload knowledge, ADR and Clay reach only about two-thirds of their peak throughput. Additionally, ADR and Clay take nearly 3× as long as MorphoSys to converge to their peak throughput, a consequence of performing design changes periodically. MorphoSys converges to its peak throughput faster than these systems because it uses every transaction as an opportunity to make physical design changes, in contrast to the periodic design changes made by ADR and Clay. DynaMast takes 5× longer than MorphoSys to reach peak throughput when initialized with a random physical design. Finally, by taking a holistic approach to distributed physical design and considering all of dynamic partitioning, replication and mastering, MorphoSys outperforms its competitors that consider only one of these aspects of physical design.

Single-master, multi-master and VoltDB, all reach their peak throughput in a short period of time as they do not change their physical designs in response to a workload. Note that the throughput of single-master is nearly identical in the two experiments, as the single-master architecture imposes a single physical design: a single site masters every partition, and each partition is replicated at all other sites. By contrast, the throughput of multi-master and VoltDB drop by 50% and 40%, respectively, when a random physical design is used compared to Schism's physical design. Throughput degrades for these systems as the frequency of distributed transactions increases in the randomized initial physical design, compared to Schism's physical design that aims to place partitions to minimize distributed transactions.

Effects of Record Size

Table 4.5: Effect of record size on physical design operations latency.

Metric **Record Size (Bytes)**	*split* Avg. Lat. (ms)	*merge* Avg. Lat. (ms)	*add_replica* Avg. Lat. (ms)	*remove_replica* Avg. Lat. (ms)	*remaster* Avg. Lat. (ms)
1	6.95 ± 1.9	19.1 ± 2.0	56.5 ± 1.2	3.07 ± 1.3	50.7 ± 9.1
10	7.05 ± 1.5	19.8 ± 2.4	58.7 ± 2.0	3.51 ± 2.3	52.9 ± 7.1
100	7.29 ± 2.1	20.3 ± 4.0	62.2 ± 2.3	4.65 ± 2.7	61.4 ± 8.6
1000	7.31 ± 1.5	21.3 ± 3.6	69.1 ± 1.3	4.96 ± 1.4	64.9 ± 10.5
Relative Change	5.17%	11.5%	22.3%	61.5%	28.0%

To understand the effect that record size has on both system performance and the physical design change operators, I experimented with the read-mostly, skewed YCSB workload. I use YCSB for this experiment, as it allows us to easily control the size of each data item. Table 4.5 shows the results of this experiment as the record sizes vary by a hefty 3 orders of magnitude, i.e., from 1 byte to 1000 bytes.

As the record sizes increase, the average latency of splitting and merging partitions remains mostly the same, with only 5% and 11% increases, respectively. The split and merge operations are not dependent on record size, because they operate on metadata of partitions; they do not need to read or write any of the records.

Adding a replica requires physically reading record data, and sending it over the network. Thus, as the data size increases, so too does the time taken to add a replica. Similarly, removing a replica partition must free the memory associated with the replica record, which results in increased latency for larger records. Finally, remastering may need to wait for the system to propagate and apply updates that take longer for larger records. Thus the latency of remastering increases as the record size increases.

4.7 Summary

In this Chapter, I presented MorphoSys, a distributed database system that automatically modifies its physical design to deliver excellent performance. MorphoSys integrates three core aspects of distributed design: grouping data into partitions, selecting a partition's master site, and locating replicated data. MorphoSys makes comprehensive design decisions using a learned cost model and efficiently executes design changes dynamically using a partition-based concurrency control and update propagation scheme. MorphoSys improves performance by up to 900× over prior approaches while precluding the use of static designs requiring prior workload information. MorphoSys ability to generate and adjust distributed physical designs on-the-fly without prior workload knowledge paves the way for the development of self-driving distributed database systems.

Chapter 5

Proteus: Adaptive Storage for Hybrid Database Workloads

In this chapter, I present how **Proteus**, a scale-out distributed DBMS, provides the benefits of both OLTP and OLAP systems in an *integral* HTAP system. Proteus *adaptively* and *selectively* stores data in *multiple formats and storage tiers* to support HTAP workloads efficiently. Proteus makes storage decisions dynamically *based on the workload* and leverages *layout-specific optimizations* that enable it to *concurrently* achieve OLTP throughput comparable to row-oriented storage systems and OLAP query latencies that are on par with column-oriented storage systems.

5.1 Adaptive Storage

As discussed in Chapter 1, neither row nor column format is optimal for an HTAP workload that contains both OLTP transactions and OLAP queries. Furthermore, a system architecture that statically replicates all data in both formats suffers from maintaining replica state efficiently and consistently [26, 144]. In contrast to these architectures, Proteus makes granular storage layout decisions, replicating when it benefits system performance. I elucidate the case for adaptive storage by introducing the set of storage decisions Proteus makes and illustrating its benefits by example.

```
insert into orderline
    (200 /* order_id */,  12 /* item_id */, 30 /* quantity */,
     55 /* amount */ , '2021/09' /* delivery */)
```

(a) New Order (OLTP)

```
update orderline set delivery = '2021/12'
where order_id = 200
```

(b) Delivery (OLTP)

```
select sum(amount)
from orderline where
    delivery between '2019/01'  and  '2020/12'
    and quantity between 1 and 100000
```

(c) Query 6 (OLAP)

```
select sum(when  i_data like 'PR' then amount
                        else 0) / sum(amount)
from orderline, item where item_id = i_id
    and delivery between '2019/01'  and  '2020/12'
```

(d) Query 14 (OLAP)

Figure 5.1: OLTP and OLAP operations from TPC-C and TPC-H benchmarks.

5.1.1 Storage Decisions

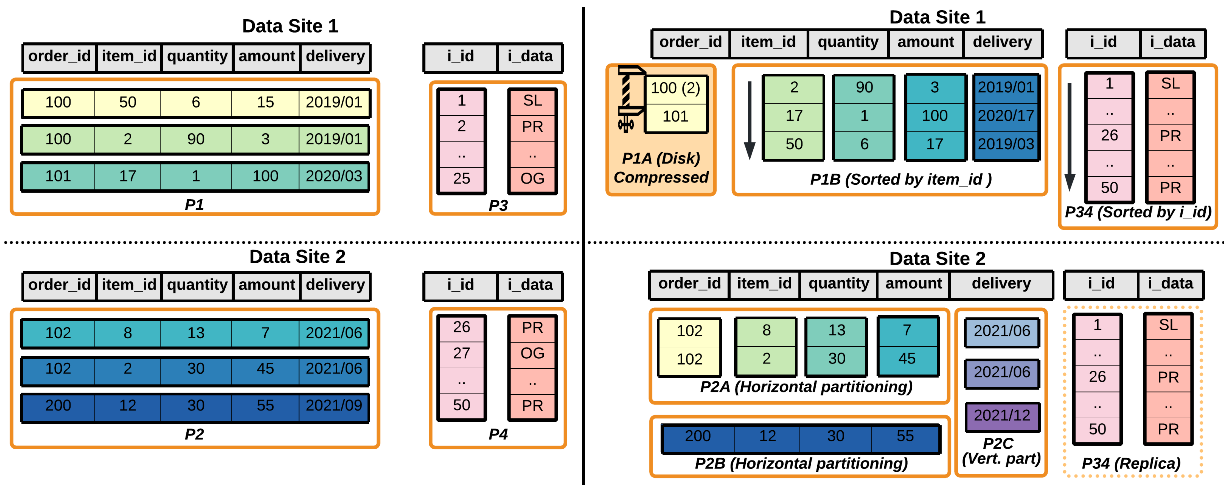
Proteus supports multiple – row or column – storage formats across multiple – memory or disk – storage tiers in the distributed system. Thus, for a given data item, Proteus adopts a *storage layout* given by its *storage format* and *tier*. Additionally, Proteus supports storage layout optimizations, such as maintaining the data in sorted order or in a compressed form [9, 89, 90, 213, 8, 10]. Because Proteus is a distributed database system, for each data item it selects a master (or primary) site where update transactions execute and a storage layout for that site. As Proteus selectively replicates data, it decides at which sites to store replicas of a data item along with an associated layout for each replica.

Proteus autonomously manages data by changing its storage layout on-the-fly as the workload executes. These changes include any or all of altering the storage format, storage tier, master and replica locations, and data partition membership. Such changes enable Proteus to adapt to the workload, as I illustrate next using a running example.

5.1.2 Example

Figure 5.2a shows a storage configuration with two data sites that store data from the *orderline* table along with the *item* table. In this initial configuration, both the *orderline* and *item* tables are partitioned across the two data sites: data partitions $P1$ and $P3$ at the first data site and partitions $P2$ and $P4$ at the second data site. The data partitions for the *orderline* table ($P1$, and $P2$) are in row format, while the partitions storing the *item* table ($P3$ and $P4$) are in column format.

Figure 5.1 shows four transactions two OLTP transactions from TPC-C [1] and two



(a) Initial State (b) Adapted State

Figure 5.2: An initial storage layout of data in Proteus, and adapted state after a series of storage layout changes.

OLAP queries from TPC-H [4]. The two OLAP queries analyze the orderline table, with the Query 6 (Figure 5.1c) finding total order amounts while Query 14 (Figure 5.1d) joining the *orderline* and *item* tables to find promotional item order amounts. The OLTP transactions insert a *new order* into the *orderline* table (Figure 5.1a), and update the *delivery* time of a recent order (Figure 5.1b).

Observe that executing *Query 6* (from Figure 5.1c) over row-oriented storage requires accessing every attribute from the relevant rows. Furthermore, the results from each site in the distributed system have to be combined to generate the final result. Observe that executing *Query 14* over this storage layout requires performing a *distributed* join of data, i.e., data in partition $P1$ (*order_id* 100) must be joined with data in partition $P4$ (*item_id* 50). Executing a distributed join requires transferring over the network all of the data necessary to perform the join, which is more expensive than transferring reduced partial (join) results to be merged [117, 59].

In contrast to Figure 5.2a, Figure 5.2b shows the *adapted* data storage layout after a sequence of layout changes made by Proteus. Relevant to *Query 14*, this adapted storage layout replicates the *item* table as $P34$ at both sites. Consequently, *Query 14* can perform its join locally, reducing the distributed execution to only the merge of partial results. Selective replication of the *item* table is an efficient choice as the table is read-only and hence does not require update maintenance [13, 201]. A further change at data site 1 is that the layout of the *orderline* table ($P1A$ and $P1B$) is columnar, which accelerates the execution of both queries *6* and *14* by reducing the amount of data accessed when executing the queries. Finally, as the chosen storage layout of both partitions $P1B$ and $P34$ are sorted by the join key (*item_id*), the system efficiently joins the data using merge-join.

In Figure 5.2b, Proteus also changes the storage layout of the *orderline* table at data site 2, fragmenting the data (originally $P2$) into three partitions ($P2A$, $P2B$, and $P2C$) via horizontal and vertical partitioning. Hence, there is not one partitioning scheme for the entire orderline table as tables are adaptively partitioned. Data in the *delivery* column is stored in a row format, as is the data associated with *order_id* 200 that was inserted as a result of the *NewOrder* operation (Figure 5.1a). Note that historical *orderline* data is stored in a column format ($P2A$). This storage layout is effective for analytical queries such as *6* and *14*, as these data are less likely to be updated. In contrast, recently (and likely to be) updated data are stored in the OLTP-oriented row format. Vertically partitioning the *delivery* column, a storage optimization also known as *row splitting* [87], reduces data contention from updates to only *delivery* time for recent orders, such as in the *Delivery* transaction (Figure 5.1b).

In summary, Proteus' adaptive storage layout reduces both OLAP and OLTP execution latency. Using cost-benefit analysis (Chapter 5.3), this latency reduction is achieved by (i) distributing and pushing processing to local data sites, (ii) using a storage format aligned with data access patterns, (iii) promoting and prioritizing data locality for in-memory processing over disk-based residency, and (iv) employing optimizations such as sorting and compression, which allows for the selection of query operators to attain efficient execution.

5.2 Proteus System Architecture

I now describe the design of Proteus with a focus on storage layouts and how Proteus adaptively manages them to execute operations over data efficiently.

5.2.1 Data Storage

Proteus stores partition data in a row or column-oriented format, and on disk or in-memory. All layouts support reads, writes and updates, as well as conversion between formats or tiers.

Row-Oriented Storage

In memory, Proteus stores each row of a partition using a fixed-size byte array, which is optimized for OLTP transactional access to many cells within a row. To determine the size of the byte array, Proteus uses the table schema and the columns contained in the relevant partition. For example in Figure 5.2a, Proteus stores each row in $P1$ in a 32 byte array: 4 bytes for each of the integer columns (*order_id* and *item_id*), and 8 bytes for each of the decimal and timestamp columns (*quantity*, *amount*, *delivery*). Proteus stores variable-sized data, such as strings, using 12 bytes with 4 bytes to encode the data size and 8 bytes to store a pointer to the data or the data itself if it fits within the 8 bytes to avoid additional memory accesses. Multi-versioning is used to support efficient updates to data stored in rows. Proteus uses the last 8 bytes in the byte array to store a pointer to a byte array storing the previous version of the row. Thus, once a row is written as a byte array, the data is read-only; updates rewrite the entire row. Proteus stores the partition's data by maintaining an array of pointers to each row's most recently stored version, updating each entry when updates occur.

To store row-oriented data on disk, Proteus divides data into two parts: an index and stored data. Each row's index entry contains an offset into where the row's data is stored. Data for each row is stored similarly to in-memory data; however, any variable-sized data is inlined directly after its length. Proteus' disk-based representation of row data allows for both point-based reads – by reading the index and corresponding offset locations – and data scans.

Proteus supports in-place updates if the update does not change the relative data size; otherwise, Proteus rewrites the entire partition's data on disk. Consequently, Proteus buffers updates in memory and applies them as a batch to disk. Consequently, Proteus buffers updates in memory and applies them as a batch to disk.

Column-Oriented Storage

To store a partition in a column-oriented format in memory, Proteus stores each column in a fixed-sized data array [108] along with two index arrays. For columns with fixed-size data, such as integers, each entry in the array corresponds to the data stored. For variable-sized data, such as strings, each array entry is stored as a length (using a fixed length of 4 bytes), followed by the bytes containing the actual data stored in its entirety. The first index array is an offset array that stores the corresponding *row_id* for each entry in the data array. The second index array is a position array that stores the offset into the data array that corresponds to the data stored for each *row_id*. These index arrays allow Proteus to efficiently locate where cell data is stored in the column for point reads. Proteus stores column data on disk using a format similar to Parquet [3], first storing the metadata including the two index arrays followed by the values.

Proteus supports flexible ordering of data: each column in the partition may store data in the same order — by *row_id* or based on a total order over the columns such as in partition $P1B$ (Figure 5.2b), or each column may be sorted independently.

Additionally, Proteus supports compressing data stored in columnar format using run-length encoding (RLE) [8] by prefixing each entry with 4 bytes to indicate the length of the run, as shown in partition $P1A$ (Figure 5.2b). Because RLE stores one value per compressed run and stores data contiguously, database operators can operate directly over compressed data. These properties have been identified as being key to allowing operating directly over compressed data [8], compared to other compression techniques such as null suppression or dictionary compression. For example, RLE allows the run length to be used when performing aggregation or outputting the results of a join (Chapter 5.2.3). RLE works best when data is already sorted or contains a high frequency of repeated elements. Finally, updating compressed data may require decompression if the stored values change, though run lengths can be updated directly.

Proteus buffers updates to column data in a *delta store*, which stores data in memory as rows in a hash-table indexed by *row_id* [106]. Thus, if scans or point-reads require accessing more recent data than stored in the columns, Proteus combines the stored column data with data from the delta store. Updates in the delta store are periodically merged with the column data to create a new version of the data for storage on disk or in memory.

Zone Maps

Independent of the storage layout, Proteus maintains zone maps [59] that maintain the minimum and maximum value for each column stored within a partition. Zone maps allow skipping data in a partition if the minimum and maximum values indicate that a predicate in a query cannot be satisfied by any data item in the partition. For example, in Figure 5.2a, the zone map of partition $P2C$ would indicate that there are no *orderline* entries that satisfy the *delivery* predicates in *Query 6* or *14*. Proteus maintains zone maps in memory and in row format, as they are of fixed size and accessed by point queries.

5.2.2 Concurrency Control and Replication

To provide SSSI, Proteus uses the partition-based dependency-tracking concurrency control algorithm introduced in Chapter 4.3.2 that tracks: (i) per-partition version numbers and (ii) the dependencies among partitions and their versions to ensure that transactions read from a consistent snapshot of data. To implement this concurrency control scheme, Proteus uses a lock per-partition to ensure that updates within a partition do not conflict but keeps multiple versions of data so that read-only operations do not conflict with updates.

When data is stored in a row format, each update creates a new copy of the row (Chapter 5.2.1). In contrast, partition data stored in a column format represents the snapshot of the partition data at a specific version, with new updates stored in the row format delta store. Consequently, read operations over column format data only merge data from the delta-store if the snapshotted column data cannot satisfy the SSSI versioning requirement.

Proteus' concurrency control scheme makes changes to the *depdendency recording* and *consistent read* rules for distributed execution. Specifically, if a transaction executes at multiple sites, sites exchange observed dependency information at the start of the transaction to produce a single begin timestamp used at all sites. This state exchange ensures that at all sites, the transaction reads the same version of replicated data. Hence, transactions continue to follow the consistent read rule. Similarly, when committing an update transaction, the transaction uses a single commit timestamp across all involved sites, which the transaction's coordinator computes between the first and second phase of the two-phase commit. Hence, all sites record the same dependency information for the transaction, ensuring that subsequent transactions observe a consistent snapshot state, independently of where they execute.

Both Proteus' concurrency control and adaptation advisor logic make use of transactional read/write information. For analytical queries, clients determine read cell ranges from the columns accessed in each table necessary to execute each query, e.g., *Query 14* (Figure 5.1d) accesses the *i_data*, and *i_id* columns in the *item* table and the *item_id*, *amount*, and *delivery* columns in the *orderline* table.

5.2.3 Transaction Execution

Given the storage layouts supported in Proteus, the system capably presents a uniform transaction execution interface. Proteus' adaptation advisor uses a query tree (Figure 5.3a) to develop a physical execution plan for each submitted transaction that specifies

Table 5.1: Proteus' cost functions and their arguments.

Cost Function	**Arguments**
Storage Layout-Aware	
Bulk Load Insert/Update/Delete Point Read	(i) # Cells accessed (ii) Column sizes
Scan w/ predicate & projection *(Sequential, Sorted, Index)*	(i) Cardinality (ii) Column sizes (input & output) (iii) Selectivity
Sort Hash	(i) Cardinality (ii) Column sizes
Join *(Hash, Nested Loop, Merge)*	(i) Cardinality (left, right, output) (ii) Column sizes (left, right, output) (iii) Join selectivity
Aggregate *(Hash, Sort)*	(i) Cardinality (input, output) (ii) Column sizes
Storage Layout-Agnostic	
Network Request	(i) CPU utilization at source & dest. (ii) # Bytes (sent & received)
Lock Acquisition	(i) Contention of partition
Waiting for Updates	(i) # Updates needed
Commit	(i) # Partitions read and written (ii) # Sites involved in transaction

where, and how, each operation should execute (Figure 5.3b). At a data site, the transaction execution layer executes the physical execution plan including accessing stored data, coordinating distributed execution with other data sites, and computing the transaction's result. To do so, Proteus applies operators such as *scan*, *join*, *update*, *insert* and *delete* that iterate over partition(s) (Table 5.1). Proteus chains these operators together to execute each transactional request. Thus, Proteus uses partition-at-a-time processing, which is a hybrid approach [97] between tuple-at-a-time and vectorized processing [100], and reduces the number of repeated accesses to data partitions.

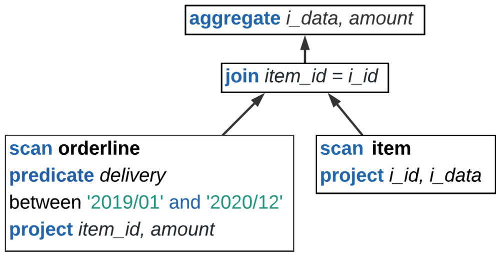
Proteus' operators come in two forms: storage-aware or storage-agnostic. Storage-agnostic implementations use the same generic storage interface regardless of how the system stores the relevant partition data. For example, acquiring a lock on a partition is agnostic to the layout. Storage agnostic data accesses and updates use cell-based operations. Storage-aware operators leverage knowledge of the storage layout to optimize execution. For example, in Figure 5.2b, partitions $P1B$ and $P34$ are sorted on the join attributes (*item_id* and *i_id*, respectively), hence a sorted column-storage aware *join* operator that implements the *merge* join algorithm leverages this knowledge to efficiently execute the join. Generally, the storage-aware operators are designed to (i) use column-specific operators, such as the invisible join algorithm for hash-joins [9], (ii) operate directly over compressed or sorted data, and (iii) use block-based accesses for data residing on disk.

Proteus uses column-specific operators for scans and column-specific operators for scans and joins. Specifically, when performing hash-joins among column partitions, Proteus uses the invisible join algorithm [9] that rewrites join conditions as predicates and evaluates these predicates using hash-lookups to determine the intersection of positions that satisfy the predicates. To perform scans, Proteus' column operators scan each column in the partition independently, recording which cells satisfy the predicate in bitmaps. Proteus merges bitmaps together and extracts the relevant tuples to produce a result set that satisfies the predicate [9].

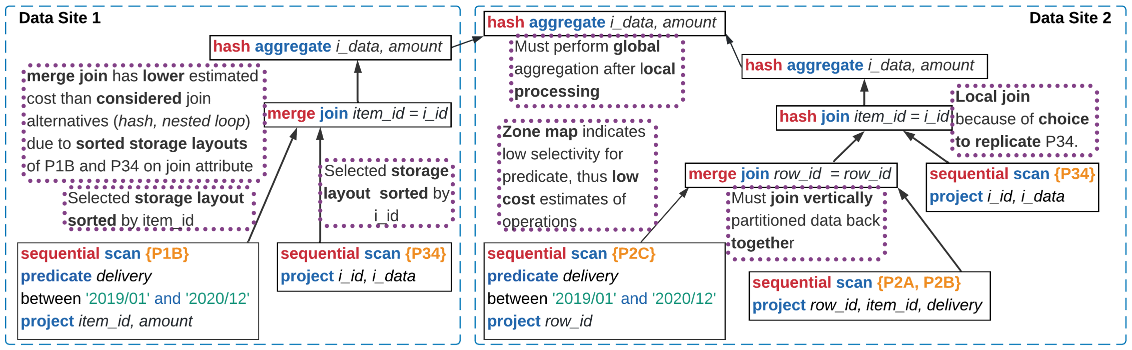
When possible, Proteus operates directly over compressed and sorted data. For example, if summing a compressed column (aggregating), Proteus can take the dot-product of each value and encoded length rather than summing each cell individually. Proteus also leverages the compressed length when performing joins to output the results of multiple tuples in one pass [8]. When searching for values that satisfy predicates, over sorted data, Proteus uses binary search to find the start or end position of values satisfying the predicate values satisfying within a partition, rather than scanning the entire partition. Proteus' adaptation advisor leverages sort order information to eliminate the need for sort operations in a physical execution plan when performing merge-joins or sort-based aggregations.

Proteus uses block-based algorithms for operations over data that reside on disk [157]. In general, these algorithms operate over a block of data at a time to reduce the number of memory pages needed to execute the query, which allows Proteus to continue to store partition data in memory without thrashing. Proteus uses block nested loop join, block hash joins and a block merge-join via external sort.

As a distributed system, Proteus coordinates transactions that access data spanning multiple sites. To execute distributed analytical queries, Proteus executes joins across sites, coordinating both data transfers and aggregation of results. For example, in Figure 5.3b, each data site executes a local join, but data site 2 coordinates the transaction by aggregating the results globally. To do so, Proteus leverages the replicated *item* table partition in the join at both data sites (Figure 5.3b). To ensure the correctness of the join result, at least one side of a join executes over precisely one copy of each partition, which in the



(a) Query Tree


Data Site 1
hash aggregate i_data, amount
merge join has lower estimated cost than considered join alternatives (hash, nested loop) due to sorted storage layouts of P1B and P34 on join attribute
merge join item_id = i_id
Selected storage layout sorted by item_id
Selected storage layout sorted by i_id
sequential scan {P1B}
predicate delivery
between '2019/01' and '2020/12'
project item_id, amount
sequential scan {P34}
project i_id, i_data
Data Site 2
hash aggregate i_data, amount
Must perform global aggregation after local processing
hash aggregate i_data, amount
Local join because of choice to replicate P34.
hash join item_id = i_id
Zone map indicates low selectivity for predicate, thus low cost estimates of operations
merge join row_id = row_id
Must join vertically partitioned data back together
sequential scan {P34}
project i_id, i_data
sequential scan {P2C}
predicate delivery
between '2019/01' and '2020/12'
project row_id
sequential scan {P2A, P2B}
project row_id, item_id, delivery

(b) Physical Execution Plan

Figure 5.3: Proteus' query tree and physical execution plan for TPC-H Query 14 (Figure 5.1d) under storage layout from Figure 5.2b.

example corresponds to partitions $\{P1B, P2A, P2B, P2C\}$ all storing the *orderline* data. Pipelined join execution over different partitions across all sites eliminates the need for duplicate generation of results. Proteus coordinates distributed updates using two-phase commit, if necessary. Since master partition placement is a storage layout decision, the adaptation advisor can adaptively change the master placement if its cost-based model determines the change to be beneficial (Chapter 5.3.4).

5.2.4 Changing Storage Layouts

Proteus supports storage layout adaptivity by changing any or all of: (i) the storage format or tier of a data partition (ii) adding or removing storage optimizations (iii) changing the data partitioning (iv) adding or removing replicas of a partition (v) changing the location of the master copy of a partition. Here, I focus on the mechanism for executing these changes. Chapter 5.3 details how the adaptation advisor makes decisions for adapting the system storage layout.

To change the storage format or tier, Proteus reads a consistent snapshot of the data into memory and bulk loads the data into the respective storage layout. For example, to bulk load row format data into memory, Proteus allocates a fixed-size buffer for every row and updates each cell as it reads the data. By contrast, bulk loading row format data that will reside on disk requires dynamic allocation of each row based on variable-sized columns, which Proteus writes to disk sequentially. Enabling sorting entails changing data storage using bulk load operations; removing a sort order does not change the data layout but ceases to maintain the sort order on subsequent updates. Compressing and decompressing stored data result in changing how it is stored, as the RLE scheme prepends the length of each run before each data item.

Changes to data partitioning schemes occur dynamically by merging or splitting partitions, either horizontally (row-wise) or vertically (column-wise). For example in Figure 5.2b, Proteus forms partition $P2C$ by vertically partitioning $P2$ in Figure 5.2a, while a subsequent horizontal partitioning forms partitions $P2A$ and $P2B$. By contrast, Proteus forms partition $P34$ (Figure 5.2b) by merging partitions $P3$ and $P4$ (Figure 5.2a). Changes to the horizontal partitioning of row formatted data simply require changing row mapping from one partition to another. A similar operation occurs for vertical partitioning of column-format data. By contrast, horizontal partitioning of column data is bulk reloaded into the new partitions.

As in Chapter 4.3.4, Proteus supports dynamically adding or removing partition replicas, and changing partition mastership.

5.3 Proteus' Adaptation Advisor

Proteus' adaptation advisor generates a physical execution plan for each client request based on the system's storage layout, and adapts its layout based on the workload to further improve system performance. The adaptation advisor accomplishes these tasks through (i) tracking the current storage layout using metadata, (ii) modelling the workload to estimate access latencies under current and adapted storage layouts, and (iii) reusing previous decisions to reduce the latency of planning. I expand on each of these techniques next.

5.3.1 Partition Metadata

The adaptation advisor tracks the metadata state of each data partition in a concurrent hash-table structure for efficient lookups. For each partition, the adaptation advisor maintains: (i) the partition bounds (minimum and maximum *row_id* and columns) (ii) the storage layout of each replica of the partition (iii) access frequencies over different time scales (minutes and hours) for updates, point reads, and scans (iv) a zone map (v) the set of partitions frequently co-accessed with the partition as a result of updates or joins. Proteus uses access frequencies to predict upcoming accesses to the partition and to estimate access costs under different storage layouts. The adaptation advisor uses the zone maps to estimate the selectivity of predicates and joins. Tracking co-access likelihood enables the adaptation advisor to reduce distributed coordination by co-locating co-accessed partitions.

The adaptation advisor also maintains per table column statistics, including each column's average size and per column access rates. Proteus uses these statistics to estimate the storage space required to store a given partition and per column access trends.

5.3.2 Predicting Data Access Latency

To select a physical execution plan or decide on a storage layout change, Proteus quantitatively evaluates the effects of its decisions. As physical operators enable layout changes, an interpretable way to capture the cost of each operator is the time it takes to execute, i.e. its latency. Using latency to compare the effects of different storage layouts on performance also serves to directly minimize transaction latency [13, 80, 175, 191]. Hence, Proteus predicts the latency of operations in the system using learned cost functions based on the statistics described in Chapter 5.3.1.

Table 5.1 summarizes the different cost functions that are used to estimate the latency of executing a query or a storage layout change. Recall that cost functions are classified as storage layout-aware, such as updating data, or layout-agnostic, such as the latency of performing a network request. Proteus learns a single cost function for storage layout-agnostic functions, while for storage layout-aware functions Proteus learns a cost function per storage layout based on the storage tier, format, and enabled optimizations.

Formally, Proteus learns a cost function F for each storage operator in Table 5.1 that consumes the arguments listed in the table.

Proteus decomposes each transaction into storage-aware operators that it chains together to execute the transaction. Proteus estimates the latency of the transaction by summing the predicted latency of each operator. For example, to predict the latency of the *Delivery* transaction (Figure 5.1b) in Figure 5.2a that updates a partition (P2) in a row layout, Proteus combines the predictions of (i) sending a network request to the data site, (ii) acquiring a lock, (iii) performing the update on the row, and (iv) committing. Each of these predictions is parameterized; for example, the latency of performing the update is parameterized by: (i) the number of cells accessed and (ii) the total average size of each cell updated.

Using storage layout-specific cost functions and parameterizing the cost functions allows Proteus to predict the latency of transactions under different physical designs. For example, consider the example of the *Delivery* transaction but now Proteus wishes to compute the latency of this transaction under a columnar layout. In this case, Proteus can replace the row-format predictor of performing an update with the column-format predictor of performing an update. Similarly, if Proteus vertically partitions the data (as in Figure 5.2b), then the parameters to the cost function are altered: as the contention on the partition decreases (an argument for acquiring the lock), and the size of stored partition data decreases (an argument for the update).

Predictors

Proteus learns the cost function F using three different learning algorithms: (i) linear regression, (ii) non-linear regression and (iii) a neural network model. As shown in Chapter 5.4.3, these algorithms have significantly different latencies for both inference (making a prediction) and training (building the model), as well as accuracy differences. Proteus uses the linear regression algorithm, which has the lowest inference latency when predicting in latency-sensitive situations such as generating an execution plan. In contrast, the non-linear regression and neural network algorithms are more accurate than the linear regression algorithm but take longer to train and converge. Hence, Proteus asynchronously updates the decision cache by averaging the predictions from all three models, so subsequent decisions take advantage of the more accurate models with lower latencies.

I now describe each of the three algorithms used in the cost functions. Each algorithm takes n arguments, as in $F(a_1, ..., a_n)$, and returns a scalar prediction y. The system also collects the true observed value for a given estimated cost, o, which is the latency of the specific operation. As an example, for an update transaction, each operator would report different observed latencies; that is, one observed latency o would correspond to the time it takes to update data. Periodically (by default every 15 seconds), with each observation o, a function's corresponding arguments $(a_1, ..., a_n)$ and prediction y are updated by Proteus via training.

Linear Regression The linear regression algorithm makes the prediction $F(a_1, ..., a_n) = y$ by learning weights $w_0, w_1, ..., w_n$ and predicting $F(a_1, ..., a_n) = w_0+(a_1 \cdot w_1)+...+(a_n \cdot w_n)$. Proteus uses stochastic gradient descent to update the weights $(w_0, w_1, ..., w_n)$ with a mean squared error loss function: $(o - y)^2$.

Non Linear Regression Proteus uses Dlib's [101] kernel recursive least squares algorithm (KRLS) [63] as its non-linear regressor. The KRLS algorithm works by learning a linear regressor (using recursive least squares) over a higher-dimensional feature space. This higher-dimension feature space is induced from the input arguments $(a_1, ..., a_n)$, observations o, and a kernel function. By default, Proteus uses the common and popular radial basis function kernel.

Neural Network Regressor To support a neural network regressor, Proteus uses Dlib's [101] multilayer perceptron that uses backpropagation to update the internal weights of the network. Dlib uses a sigmoid activation function at each node, so the network produces an output between 0 and 1. Hence, Proteus scales the final output from the network to between 0 and a predefined maximum latency (30 seconds). By default, neural networks used for regression in Proteus have two hidden layers, allowing the model to learn arbitrary functions. For a cost function $F(a_1, ..., a_n)$ with n arguments, the input layer has $n + 1$ nodes, an output layer with 1 node, and two hidden layers each have $\frac{n+2}{2}$ nodes, which follows the principle of averaging the number of input and output nodes.

Collecting Observations

Data sites collect observed latencies of database operators and report these observations to Proteus to train the cost functions. To collect these observed latencies, the data sites utilize an observation collection API that records the start and end times and model arguments of each invocation of a storage operator. The storage operator is also responsible for computing the input arguments associated with the operator. In the update transaction example, the update operator invokes the API to record the start time of its execution, the end time of its execution, reporting the number of cells written, and the size of the columns of data read and written. The data site adds this to a per-thread observation data structure, which stores a list of observed latencies and their arguments for each storage layout and operator pair. The data sites report observations to Proteus by swapping each thread's observations with an empty set and merging the observations together. The data sites report these merged observations to Proteus, which uses them as data to train its cost functions.

5.3.3 Estimating Data Access Arrivals

Proteus considers the effect of adapting storage layouts on future requests. Recall from Chapter 1 that both analytical and transactional workloads exhibit temporal and cyclic trends in requests that arise due to follow-the-sun behaviour, or scheduled reporting [175, 121, 190]. Hence, Proteus uses learned models to predict the likelihood of future data accesses and their arrival time. Doing so requires predicting the number of partition accesses by type (e.g., scan versus update) in a given upcoming time window.

Predicting when data will be accessed and the volume of accesses entails: (i) predicting trends at different temporal granularities, such as daily, weekly or yearly patterns, (ii) handling growth and spikes in requests, which can occur on specific dates such as the fiscal year-end, and (iii) adapting to changes in workloads over time, which may arise if the submitted queries or transactions change due to user needs or preferences [193, 121]. Thus, Proteus uses predictive models with both a periodic component to capture the long-term *periodic trend*, and the local trend to capture *short term* transient effects such as spikes in the number of requests.

Proteus tracks accesses on a per-partition basis by access type (update, point read, or scan). These statistics are tracked over 5-minute intervals for the past day and hourly for a month and supplied as input to Proteus' predictive models.

Predictors

Formally, Proteus predicts $\delta(T, \tau)$ that represents the number of requests of type T that will arrive in time window τ. Proteus supports two different predictors to estimate $\delta(T, \tau)$: SPAR and a hybrid-ensemble. Both predictors combine a *periodic trend* prediction, that is, how access patterns change over a recurring period (e.g., accesses follow an hourly cycle), and a *short term* effect component that considers the short term trend in accesses (e.g., accesses are becoming more common over an hour). For SPAR, the period is user-defined, while the hybrid-ensemble learns the period.

For both predictors, Proteus considers τ in five-minute intervals over a day. For notational simplicity, if τ is less than the current time (that is, in the past), I define $\delta(T, \tau)$ as the actual number of accesses to T in the time window represented by τ. I describe how Proteus efficiently collects this access history in Chapter 5.3.3.

SPAR Proteus uses SPAR [43] to combine long-term *periodic trends* with *short term* effects. To learn the *periodic trend*, SPAR requires a user-defined period Ψ, and a number of periods to examine Ψ_n. SPAR considers the number of requests to T in previous time intervals (represented by $\delta(T, \tau - i\Psi)$ for $1 \leq i \leq \Psi_n$). Proteus weights these previous numbers of requests by coeffecients b_i as shown in Equation 5.1. Coeffecients are learned using linear regression.

$$\sum_{i=1}^{\Psi_n} (b_i \cdot \delta(T, \tau - i\Psi)) \tag{5.1}$$

Proteus also uses SPAR to considers how *short term* access counts shift within the period Ψ compared to the access counts in previous periods. SPAR averages this access count, as $\gamma(T, \tau)$ in Equation 5.2.

$$\gamma(T, \tau) = \delta(T, \tau) - \frac{1}{\Psi_n} \sum_{i=1}^{\Psi_n} \delta(T, \tau - i\Psi) \tag{5.2}$$

To compute the *short term* shift within the period Proteus uses SPAR to weight $\gamma(T, \tau - j)$ using learned coefficients c_j for the j intervals within the period Ψ. There are Ψ_k such intervals within a period. Consequently, Proteus computes the short term shift in access counts as shown in Equation 5.3.[1]

$$\sum_{j=1}^{\Psi_k} (c_j \cdot \gamma(T, \tau - j)) \tag{5.3}$$

Proteus combines the *periodic* prediction (Equation 5.1) with the *short term* trends (Equation 5.3), via a sum to predict $\delta(T, \tau)$, as shown in Equation 5.4.

[1]By default, Proteus uses a period of one hour, and considers the previous day; hence $\Psi_n = 24$. As intervals are 5 minutes long, $\Psi_k = 12$.

$$\delta(T,\tau) = \sum_{i=1}^{\Psi_n}(b_i \cdot \delta(T,\tau - i\Psi)) + \sum_{j=1}^{\Psi_k}(c_j \cdot \gamma(T,\tau - j)) \tag{5.4}$$

Hybrid-Ensemble Proteus' hybrid-ensemble (HE) predictor combines a periodic predictor using a recurrent neural network (RNN) to capture the *periodic trend*, and a *short term* predictor using linear regression. Importantly, the HE predictor does not require used-defined periods as the RNN learns the periodic trend.

Proteus' linear regression predictor $\delta_L(T,\tau)$ predicts the number of accesses to T at time τ based on past access counts over a sliding window. As the regression is linear, it captures only the *short term* access trend. That is, it captures whether accesses are increasing or decreasing over time, and if so, by how much.

Proteus' RNN predictor $\delta_R(T,\tau)$ captures *periodic trends* without requiring prior knowledge of the period. RNNs are a class of networks where nodes in the network have cycles [200], and Proteus uses the long short-term memory (LSTM) architecture [81] for its RNNs. This architecture allows the RNN to remember values over arbitrary time intervals, which allows the RNN to learn the period. Proteus uses libtorch's [149] implementation of LSTMs with five internal layers determined empirically.

Proteus also accepts a user-defined custom holiday list that allows an administrator to define periods where additional access counts should be added or removed following a Gaussian function [193]. The holiday list allows Proteus to account for events that repeat but not in a periodic cycle, such as Black Friday, which is not on the same date every year. For each holiday, the user defines the time over which the holiday occurs (range between h_s and h_e), and parameters for the Gaussian function (peak value α, width σ^2, and centre between h_s and h_e) that capture the access counts for that time as $\delta_H(T,\tau)$. If τ is within the defined time of the holiday (i.e., $h_s \leq \tau \leq h_e$) then Proteus computes $\delta_H(T,\tau)$, given in Equation 5.5; otherwise, $\delta_H(T,\tau) = 0$.

$$\delta_H(T,\tau) = \alpha \cdot \exp\left(-\frac{\left(\tau - \frac{h_e+h_s}{2}\right)^2}{2\sigma^2}\right) \tag{5.5}$$

Proteus combines the three components of the HE to produce a final prediction $\delta(T,\tau)$ from the average of the linear trend and periodic behaviour, and any holiday adjustments as shown in Equation 5.6

$$\delta(T,\tau) = \frac{\delta_L(T,\tau) + \delta_R(T,\tau)}{2} + \delta_H(T,\tau) \tag{5.6}$$

Collecting Access History

To predict the number of accesses to a partition by type in a time window requires collecting access history to train predictive models. The number of accesses to a partition by type within a one-minute window (by default) is recorded by each data site. As the access counts are kept on a per-partition basis, these counters have low levels of contention. Proteus polls each data site to collect this information and aggregates the accesses across data sites within five-minute windows (by default). Using these newly collected access patterns, Proteus updates its access estimators via training. In the case of SPAR, the collected observations must be stored until they will no longer be needed for inference, that is, Ψ_n periods have passed. Proteus stores these access counts using a circular buffer, and new access counts overwrite prior access histories. For the hybrid-ensemble, the collected observations no longer need to be stored once the estimator has been updated; however, Proteus keeps a sampled reservoir of access histories for use when updating the estimator so that the model has access to long-term history.

Generalizability of Predictions

Finally, although not the focus of this thesis, the predictive techniques used by Proteus are generalizable to other database systems. I have demonstrated this generalizability by integrating Proteus' predictive capabilities into two systems [15]. First, an OLAP DBMS that performs cracking [74], a process that incrementally adapts indices to the workload by reorganizing data based on accesses. By integrating Proteus' capabilities, I enable the DBMS to perform cracking predictively: if Proteus predicts an area of data will be accessed in the future, it reorganizes (cracks) the area of data, thereby reducing the latency of queries that access that data. Second, I also integrated Proteus' predictive capabilities into PostgreSQL to automatically add and remove secondary indexes based on the predicted workload.

5.3.4 Query and Storage Layout Planning

The adaptation advisor generates physical execution plans based on its workload models and the current storage layout. Next, I describe how Proteus generates physical execution plans, makes storage layout decisions, and reuses past decisions to accelerate these processes.

Physical Execution Plans

Proteus begins the physical execution plan generation process by starting out with a query tree for the request, which is generated using PostgreSQL's parser and analyzer. Using the query tree, Proteus generates a physical execution plan by (i) replacing accessed tables in leaf nodes with the relevant corresponding partitions at a specific site, (ii) instantiating internal nodes with operators, and (iii) adding additional operators to handle distributed operations as necessary.

Note that in Figure 5.3a, the leaves of the query tree are accesses to the *orderline* and *item* tables. When generating the associated physical execution plan shown in Figure 5.3b, Proteus replaces these leaves with accesses to the relevant partition data identified by partition metadata. Figure 5.2b shows that with respect to the *orderline* table, there are four relevant partitions: $P1B, P2A, P2B$, $P2C$. Thus, Proteus replaces the leaf with an instantiation of access to partition $P1B$. Due to vertical partitioning of the data, accesses to $P2A, P2B$, and $P2C$ result in rewriting of the leaf nodes with separate accesses to the partitions. Proteus inserts an internal *join* node to make the required columns available together to the parent operator. Proteus also selects the sites where data access to the partition will occur, enforcing that at least one side of any join operation executes at exactly one site for each relevant partition.

Given an assignment of the site and data access to each leaf node in the query tree, Proteus assigns a physical operator to each node in the query tree. For example, Proteus selects to join partitions $P1B$ and $P34$ using a merge-join algorithm over other alternatives such as a hash or nested-loop join algorithm because its cost functions estimate the merge-join to have the lowest cost. Proteus makes this decision greedily to effectively reduce the physical execution plan search space. Operator(s) to the plan are added if a selected physical operator needs them, such as inserting sort operators before merge-join.

Proteus adds nodes to combine and coordinate distributed operations, e.g., Figure 5.3b's distributed aggregation at data site 2.

Layout Changes

Proteus plans layout changes in response to three stimuli: (i) generating a physical execution plan for a request (ii) requests predicted to arrive in an upcoming 10-minute (configurable) interval (iii) when a data site nears storage constraints, e.g., memory or disk capacity limits. In each case, Proteus integrates the workload models into one equation, the net benefit of a change defined by $N(S)$ that quantifies how beneficial a change is to the system overall. Proteus computes $N(S)$ by estimating (i) the upfront cost $U(S)$ to execute the storage layout change based on the operations needed to perform the change (via the cost functions in Table 5.1), and (ii) the expected cost effect $E(S)$ on requests predicted to arrive and on requests currently executing $(C(S))$. Proteus computes $E(S)$ and $C(S)$ by computing the difference between predicted transaction latency under the current and proposed layouts, weighted by the likelihood of the transaction executing. The adaptation advisor computes the net benefit of adaptation $N(S)$ as shown in Equation 5.7.

$$N(S) = \lambda(E(S) + C(S)) - U(S) \tag{5.7}$$

In the running example from Chapter 5.1.2 (Figure 5.2b), Proteus chose to vertically partition $P2$ to produce partition $P2C$ to mitigate the contention effects of OLTP operations, such as the delivery transaction (Figure 5.1b). The adaptation advisor estimated the proposed layout will reduce transaction latency, resulting in a positive $E(S)$. To perform this vertical partitioning necessitates an upfront cost $(U(S))$, primarily induced by the cost of scanning the original partition and bulk loading the new partitions. This change affects other queries, such as the join in Query 14, slightly increasing its latency under the proposed change as it induces another join (Figure 5.2b). However, Proteus estimates the impact is small due to the selectivity of predicates over $P2C$, resulting in a low join cost and a small negative $C(S)$. Thus, $N(S)$ is positive so Proteus proceeds with the change.

After generating a physical execution plan, the adaptation advisor performs a top-down search to find the leaf that contributes the highest data access cost to the overall plan. In the running example, the latency induced by contention when accessing partition $P2$ is the highest leaf cost, and thus the adaptation advisor considers it a candidate for a storage layout change, in this case vertical partitioning. The adaptation advisor considers if any storage layout changes that affect the high-cost leaf induce a positive net benefit $(N(S))$. If so, Proteus initiates the storage layout change and updates its physical execution plan. Proteus repeats this process, stopping when further changes do not result in a positive increase to the value of $N(S)$. In the running example, vertically partitioning $P2$ to produce $P2C$ is beneficial, so it is added to the plan. Proteus horizontally partitions $P2$ to produce $P2A$ and $P2B$ but stops after this change as further changes do not improve $N(S)$.

To make effective changes without sacrificing efficiency, Proteus tracks the average estimated leaf access cost per type of operation, normalized by the number of cells accessed, and performs this planning step (i) if this normalized cost is above average or (ii) probabilistically with probability inversely proportional to the normalized cost. Thus, Proteus reduces cost by changing the storage layouts of partitions with high access costs to generate higher potential latency savings.

Proteus periodically predicts upcoming accesses and considers storage layout changes if the predicted access pattern differs from the recent pattern of accesses. For example, if Proteus predicts frequent updates to a partition because of a cyclical access pattern but the partition has been infrequently updated recently, then Proteus adapts the partition storage layout. To do so, a physical execution plan is generated for a placeholder transaction that accesses the partition. Proteus then predictively plans storage layout changes using the aforementioned procedure. The placeholder transaction is instantiated with accesses based on recorded partition co-access statistics (Chapter 5.3.1). Proteus constrains the search space of potential predictive layout changes by considering the workload in only the upcoming 10-minute (configurable) interval. As I show experimentally in Chapter 5.4.3, Proteus executes layout change plans efficiently, in sub-second latencies on average, and hence can execute many changes predictively within this time window.

Data sites track their storage usage per tier and report to the adaptation advisor as they approach the capacity limit within a tier. In response, Proteus considers executing storage layout changes that reduce the consumed storage capacity by: (i) removing replica partitions (ii) changing partition mastership to another site (iii) compressing partition data (iv) moving partition data to a lower storage tier. Proteus estimates the expected benefit of each of these options for data partitions stored in the tier and selects the option that maximizes the expected benefit $N(S)$. To avoid considering all partitions, data partitions are grouped into tiers by their access statistics, including estimated access arrival times and estimated $N(S)$ for each decision for a partition group as a whole. Once Proteus selects a partition group and makes a decision, each partition within the group is repetitively considered under the decision, executing storage layout changes until the site is under its capacity limit(s).

Computing Net Benefit

Previously, I introduced how Proteus computes the net benefit of a storage layout change S, as $N(S)$ in Equation 5.7. I now formalize the computation of $N(S)$.

Table 5.2 summarizes each storage layout change in Proteus and the cost functions that the adaptation advisor combines to estimate the upfront cost of performing the storage layout change. For example, in Figure 5.2b to change the storage format of partition

Table 5.2: The upfront costs of different storage layout changes in Proteus, which are computed by combining different cost functions.

Storage Layout Changes	**Cost Function**						
	Bulk Load	*Scan*	*Sort*	*Network Request*	*Lock Acquisition*	*Waiting for Updates*	*Commit*
Change Format	✓	✓		✓	✓		
Change Tier	✓	✓		✓	✓		
Sort	✓	✓	✓	✓	✓		
Compress	✓	✓		✓	✓		
Change Partitioning (Horizontal in Row) (Vertical in Column)				✓	✓		✓
Change Partitioning (Generic)	✓	✓	✓ *(As needed)*	✓	✓		✓
Add Replica	✓	✓		✓ *(Sourc & Dest.)*	✓ *(Dest.)*	✓	
Remove Replica				✓			
Change Master				✓ *(Source & Dest.)*	✓ *(Source & Dest.)*	✓	✓ *(Source & Dest.)*

$P1B$ from a row to column format requires: (i) the adaptation advisor making a network request to data site 2, (ii) reading all the data from the row formatted partition via a scan, and (iii) bulk loading the data into a column format. The adaptation advisor uses the layout-specific cost functions, including row-specific scan and column-specific bulk loader estimators. Recall from Chapter 5.2.4 that horizontal partitioning of rows, or vertical partitioning of columns, does not need to perform full data scans or bulk data loading, simply the reassignment of pointers. Consequently, Proteus differentiates the upfront cost for these changes to data partitions from the generic change of partitions.

To compute the expected effect of a storage layout change S on a request T, Proteus estimates the latency of the request under the current layout $L_{current}(S,T)$ and the latency of the request under the adapted layout $L_{adapt}(S,T)$. Proteus estimates L_{adapt} by using different storage layout aware cost functions — in the example for partition $P1B$ (Figure 5.2b), the adaptation advisor replaces the row format scan cost function with the sorted column scan cost function. Alternatively, storage layout changes may alter the input arguments to the same function; for example, the vertical partitioning of partition $P2C$

Table 5.3: The expected change in transaction execution latency in Proteus, per cost function for each storage layout change.

Storage Layout Changes	**Cost Function**										
	Bulk Load	*Insert/ Update/ Delete*	*Point Read*	*Scan*	*Sort/ Hash*	*Join*	*Aggregate*	*Network Request*	*Lock Acquisition*	*Waiting for Updates*	*Commit*
Change Format	✓	✓	✓	✓	✓	✓	✓				
Change Tier	✓	✓	✓	✓	✓	✓	✓				
Sort	✓	✓	✓	✓	✓	✓	✓				
Compress	✓	✓	✓	✓	✓	✓	✓				
Change Partitioning	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓
Change Replication		✓	✓	✓	✓	✓	✓		✓	✓	
Change Master								✓			✓

lowers the estimated contention of the partition, an argument for the lock acquisition cost function. Table 5.3 summarizes the effect each storage layout change has on estimated transaction latency for transactions that access the changed data partitions.

Proteus weighs the estimated effect of the storage layout change on T by the likelihood of T's request arriving ($Pr(T)$), and the time to T's arrival ($\Delta(T)$). $Pr(T)$ and $\Delta(T)$ are derived from the prediction of $\delta(T,\tau)$, which is the number of requests of type T that will arrive in time window τ (Chapter 5.3.3). Given $\delta(T,\tau)$, Proteus computes $\Delta(T)$ at time t as $\tau - t$ for the minimum $\tau > t$ such that $\delta(T,\tau) > 1$, that is the first time in the future that Proteus estimates at least one request will arrive. Given such a τ, Proteus estimates the probability of such an arrival as $Pr(T)$ as shown in Equation 5.8, which takes the complement of a Poisson estimate of the probability that no requests will arrive at time τ.

$$Pr(T) = 1 - \exp\left(-\delta(T,\tau)\right) \tag{5.8}$$

Given these predictions, Proteus combines the estimated effect of the storage layout change S on T as $E(S,T)$, as defined in Equation 5.9.

$$E(S,T) = \left(L_{current}(S,T) - L_{adapt}(S,T)\right) \cdot \frac{Pr(T)}{(\Delta(T)+1)} \tag{5.9}$$

Observe that $E(S,T)$ is positive if $L_{current}(S,T) > L_{adapt}(S,T)$, which indicates that Proteus expects the storage layout change to reduce the latency of executing the request.

However, the magnitude of $E(S,T)$ is determined by: (i) the relative change in execution latency, (ii) how likely the request is to arrive, and (iii) the estimated time to request arrival.

Proteus computes $E(S)$ and $C(S)$, by summing $E(S,T)$ for each request that are ongoing (in which case $\Delta(T) = 0$ and $Pr(T) = 1$), or predicted to arrive. Observe that $E(S,T) \approx 0$ if $L_{current}(S,T) \approx L_{adapt}(S,T)$, $Pr(T) \approx 0$, or $\Delta(T)$ is sufficiently large. Consequently, Proteus restricts the set of requests that it considers to those that access data affected by the storage layout change, or are likely to arrive in an upcoming window.

Finally, Proteus combines $E(S)$, $C(S)$, and $U(S)$ into one equation that defines the net benefit of the storage layout change, as shown in Equation 5.7 as $N(S)$. Observe that $\lambda > 0$ controls the importance of the expected benefit of the storage layout change, compared to the upfront costs to perform the layout change. Note that if $N(S)$ is greater than 0, the storage layout change is considered beneficial.

Plan Reuse

To reduce planning overhead, Proteus caches previously used physical execution plans for reuse. A plan is reused in its entirety if the current storage layout satisfies the layout used in the cached plan. For example, consider Query 14 executed on the storage layout shown in Figure 5.2b using the physical execution plan shown in Figure 5.3b. If Proteus receives this same request again with the same storage layout, then it would reuse that same physical execution plan. As Proteus adaptively changes its storage layout, a single change invalidates a plan. Hence, Proteus carries out plan decision reuse. To do so, the input arguments for each operator are bucketed and the decisions made given these arguments are cached. If a subsequent decision has similar inputs, then Proteus reuses the decision with the lowest estimated cost. Proteus also uses this technique to reuse decisions for storage layout changes.

5.4 Experimental Evaluation

I now present an experimental evaluation to demonstrate the effectiveness of Proteus' storage layout adaption and how it significantly boosts HTAP system performance for varying HTAP mixes, access patterns and load.

5.4.1 Methodology and Benchmarks

The experiments are conducted on up to 18 data sites (default 6 sites) following the same configuration as described in Chapter 3.5.2.

I conduct experiments using three HTAP workloads: the **CH-benCHmark** (CH) [46], transactional **YCSB** [48] and **Twitter** [58] benchmarks. CH consists of the TPC-C OLTP [1] and TPC-H OLAP workloads [4]. The transactional YCSB workload consists of two types of transactions: a 10-key (multi-key) read-modify write OLTP transaction and an OLAP query that scans 500,000 rows, evaluates a predicate, and aggregates the results. As described in Chapter 4.6.2, the Twitter workload models a social networking application. The workload contains six OLAP transactions and three OLTP transactions with four transactions added from the Twitter API [7] that update followers, get tweets from followers, get tweets within a timespan, and get tweets starting with specific text, introducing more OLAP and OLTP transactions.

For all workloads, I follow CH's model of a client submitting either OLTP or OLAP transactions at any one time. As in prior work [46], I experimented with three mixes: *OLTP heavy* (90% of clients submit OLTP transactions), *balanced* (50% OLTP), and *OLAP heavy* (10% OLTP).

To conduct workload execution time experiments, I fixed the number of transactions executed by each client. For YCSB, each client issues 1 OLAP transaction followed by a proportional number of OLTP transactions by mix (10 for OLTP heavy, 6 for balanced, and 3 for OLAP heavy) 10,000 times. For CH, each client issues 220 TPC-H queries and the OLTP to OLAP proportions are 999:1 (OLTP heavy), 99:1 (balanced), and 19:1 (OLAP heavy). For Twitter, each client issues 300 OLAP queries and the OLTP to OLAP proportions are 1000:1 (OLTP heavy), 100:1 (balanced) and 10:1 (OLAP heavy).[2]

The YCSB database (50 GB) consists of 50 million rows and 10 columns each storing 100 bytes. The YCSB workload uses skewed OLTP accesses to generate contention and load imbalance. CH uses a scale factor of 100 (100 GB) while Twitter stores 10 million user accounts (80 GB).

Additionally, I use two real-world traces of access counts aggregated by minute: **Wikipedia** [195] and **Azure** BLOBs [166]. The Azure trace derives from a function-as-a-service workload and consists of many different client applications accessing data, which results in higher variability in access patterns compared to the more homogeneous Wikipedia trace.

[2]Ratios represent the proportion of executed OLTP to OLAP transactions.

5.4.2 Evaluated Systems

I evaluate Proteus against the alternative distributed HTAP database system architectures of **Janus** [25] and **TiDB** [84], as well as a **row**-oriented distributed database (row store or **RS**) targeted to optimize OLTP workload execution and a **column**-oriented distributed database (column store or **CS**) designed to optimize OLAP workload processing. To ensure an apples-to-apples comparison of techniques, I implemented the RS and CS architectures in Proteus, which provides for an in situ comparison of the two specialized storage formats. I implemented Janus in Proteus with Proteus' adaptive features disabled. I used the commercial open source version of TiDB. TiDB and Janus fully replicate data between their OLTP-optimized and OLAP-optimized stores. Janus executes OLTP transactions on the RS and OLAP transactions on the CS [25], and updates are propagated lazily from the OLTP to the OLAP store. By contrast, TiDB uses Raft as its replication algorithm and a cost model to determine where to run a given transaction [84].

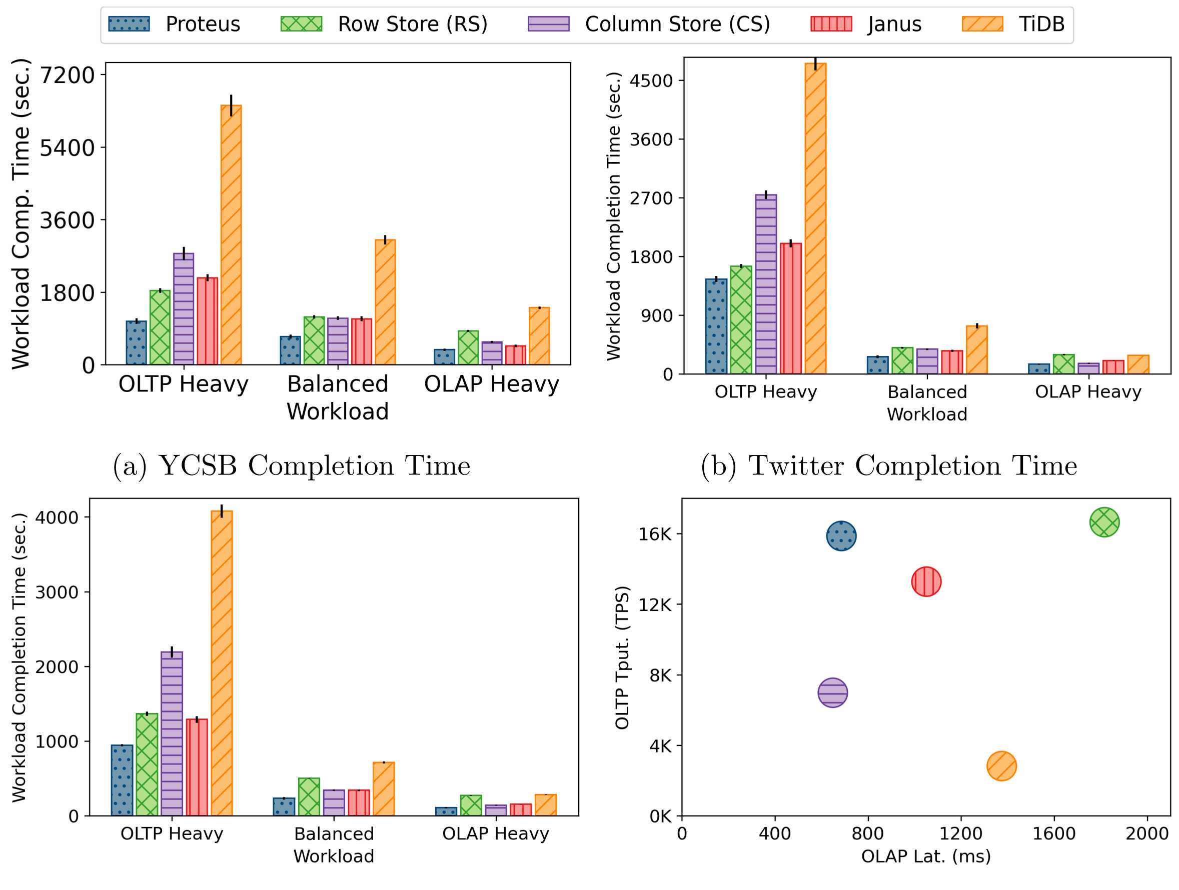
I advantaged each comparison system implemented in Proteus with an optimized replication and partitioning scheme using Schism [53] that uses *a priori* knowledge of the workload, including whether a table is read-only. These systems use the Least-Recently-Used (LRU) scheme to determine storage tier placement as LRU is appropriate for both skewed and uniform access patterns and is a popularly-used storage tier policy [141, 79].

5.4.3 Experimental Results

I now discuss Proteus' experimental results for varying HTAP mixes, access patterns and load.

Workload Execution/Completion Time

To establish Proteus' ability to process HTAP workloads efficiently, I evaluate all systems with the 3 YCSB HTAP workload mixes. I measured the (execution) time to complete these workloads, shown in Figure 5.4a. Proteus executes the balanced workload 4.4× and 1.6× faster than TiDB and Janus, respectively, and completes the workload faster than all competitors for all mixes. Proteus achieves these results because its adaptive storage techniques result in superior OLTP throughput compared to RS and OLAP latency that is competitive with CS (Figure 5.5). Only Proteus achieves this high level of performance for both OLTP and OLAP workloads. Proteus' superiority on hybrid workloads is also evident for CH (Figure 5.4c). Proteus reduces the time to execute the balanced workload



(a) YCSB Completion Time

(b) Twitter Completion Time

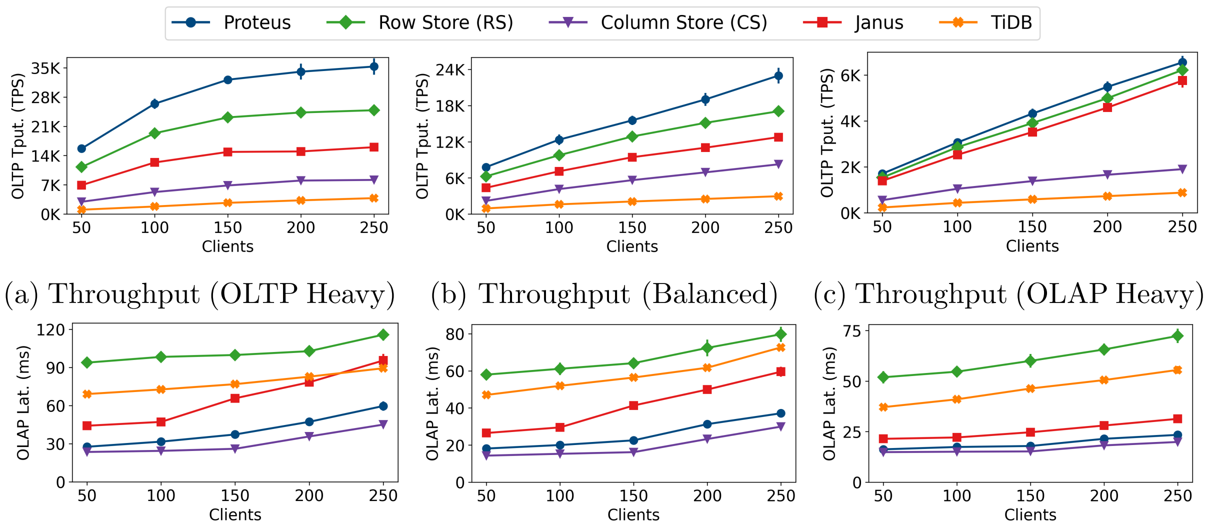
(c) CH Completion Time

(d) CH Latency vs. Throughput

Figure 5.4: Proteus executes hybrid workloads faster than all competitors for all YCSB, CH-benCHmark (CH) and Twitter mixes.

by more than 33% compared to CS, 32% compared to Janus, more than 50% compared to RS, and 70% compared to TiDB. In all mixes, Proteus executes the workload faster than all of its competitors because its OLAP performance remains competitive with that of CS while delivering equivalent OLTP performance to that of RS, as shown in Figure 5.4d.

In the Twitter workload, Proteus has the lowest workload completion time for all mixes, and reduces the time to execute the balanced workload by more than 25% compared to Janus, 30% compared to CS, 45% compared to RS, and 65% compared to TIDB. Similar to the CH workload, Proteus is superior to its competitors in the Twitter workload because its OLAP latency is similar to that of CS while delivering OLTP throughput competitive



(a) Throughput (OLTP Heavy) (b) Throughput (Balanced) (c) Throughput (OLAP Heavy)

(d) Latency (OLTP Heavy) (e) Latency (Balanced) (f) Latency (OLAP Heavy)

Figure 5.5: YCSB benchmark results over three HTAP mixes (OLTP heavy, balanced and OLAP heavy) showing OLTP throughput (5.5a–5.5c) and OLAP latency (5.5d–5.5f).

to that of RS (Figure 5.8).

YCSB

I now present Proteus' performance for the dimensions of OLTP throughput and OLAP latency. Figures 5.5a, 5.5b and 5.5c show throughput of the OLTP operations for each system on the three HTAP mixes. Proteus achieves the highest throughput, outperforming competitor systems by between 9.3× and 1.4× in the OLTP heavy workload. In this experiment, Proteus' closest competitors are RS and Janus, which execute the OLTP operations on row-oriented storage.

Examination of Proteus' storage layouts (Table 5.4 in the balanced workload) shows that the adaptation advisor learns an appropriate storage layout. Different data regions have different access patterns because the YCSB workload contains skewed OLTP accesses and uniform OLAP accesses. Proteus stores frequently updated data in row format with infrequently updated data stored in column format. Observe that the regions of data with OLTP heavy or OLTP mostly access patterns comprise 9% of the data but account for nearly half (49%) of all OLTP accesses. Proteus forms many small partitions kept in memory and a row format for data in these access groups. Consequently, most updates

Table 5.4: A description of the storage layouts chosen by Proteus for the balanced YCSB workload. The OLTP accesses are skewed; hence a significant portion of OLTP accesses access a small amount of data, while OLAP accesses are uniformly distributed among data.

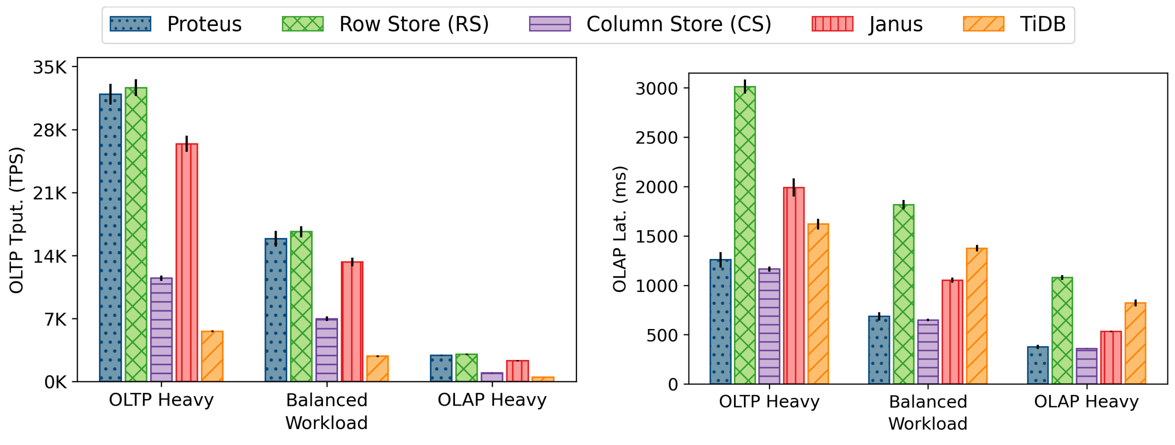
Storage Layout Attributes	**Data Access Pattern Group**				
	OLTP Heavy	*OLTP Mostly*	*Balanced*	*OLAP Mostly*	*OLAP Heavy*
Fraction of Data	2.7%	6.3%	42%	34%	15%
Portion of OLTP Accesses	34%	15%	42%	6.3%	2.7%
Avg. Number of Copies	2.25 ± 0.18	2.05 ± 0.13	1.95 ± 0.07	1.01	1.00
Master Format	Row	Row	Row	Sorted& Compressed Columns	Sorted& Compressed Columns
Replica Formats	Row	Row	Columns		
Storage Tier	Memory	Memory	Memory	Memory or Disk	Disk
Partition Size (Rows)	78 ± 12	312 ± 23	2427 ± 6	4921 ± 70	10251 ± 97
Partition Size (Columns)	5	5	10	10	10
Number of Partitions	34615 ± 689	20192 ± 472	4183 ± 24	861 ± 5	95 ± 2

execute over data stored in a row format and in memory.

Proteus' adaptive storage techniques, including vertical and horizontal partitioning of data (Chapter 5.2.4) to mitigate contention effects within and across rows, result in Proteus outperforming both RS and Janus that use static storage layouts. As the workload becomes more OLAP heavy (Figure 5.5c), Proteus adapts its storage design, trading off OLTP performance in favour of OLAP performance. I later discuss the effectiveness of vertical and horizontal partitioning techniques via an ablation study.

Figures 5.5d, 5.5e and 5.5f show the average latency of the OLAP operations for each system on the three HTAP mixes. Proteus achieves an average OLAP latency on par (within 10ms) with CS and significantly reduces the OLAP latency compared to all other competitors by between 3.1× and 1.3×.

Observe in Table 5.4 that the OLAP mostly and OLAP heavy access groups account for nearly half (49%) of the data, but just 9% of all OLTP accesses. Proteus forms large partitions for these access groups, comprising thousands of rows, in a sorted and compressed column format on disk. Consequently, as the workload becomes OLAP heavy, Proteus shrinks the latency gap from CS and achieves comparable latency to the CS because Proteus stores most of the data in a column-only format to support uniform data scan accesses across the table. Consequently, Proteus need only execute OLAP operations across data stored in a row-only format for the update-heavy parts of the database. Importantly, Proteus achieves these results while delivering more than 4× the OLTP throughput



(a) TPC-C Throughput (b) TPC-H Query Latency

Figure 5.6: CH-benCHmark (CH) results showing OLTP throughput and OLAP latency for three HTAP mixes.

of CS. Proteus further reduces space consumption by adaptively employing compression on the most infrequently updated data in the system. Moreover, this read-heavy data often resides on disk due to storage constraints, but Proteus' use of block-based join algorithms (Chapter 5.2.3) can operate over it efficiently without thrashing data that resides in memory.

Janus and TiDB replicate all data twice, consuming on average 1.33× more space than Proteus and imposing memory constraints on the system. By contrast, as shown in Table 5.4, Proteus selectively and judiciously replicates partition data into both a row and column format when there are roughly equal OLTP and OLAP accesses, reducing the amount of stored data in the system. In all, Proteus stores data in about 60,000 partitions, storing an average of 1.5 copies of each data item.

CH-benCHmark

Next, I evaluate Proteus using the CH-benCHmark (CH), an HTAP workload derived from 22 TPC-H OLAP queries and 5 TPC-C OLTP transactions. For all mixes, Proteus achieves throughput comparable with the top-performing OLTP system – within 5% of RS OLTP throughput (Figure 5.6a) *and* within 8% of the CS OLAP latency (Figure 5.6b). Neither RS nor CS can achieve anywhere near this combined high performance for *both* OLTP throughput and OLAP latency. These results demonstrate that Proteus' adaptive

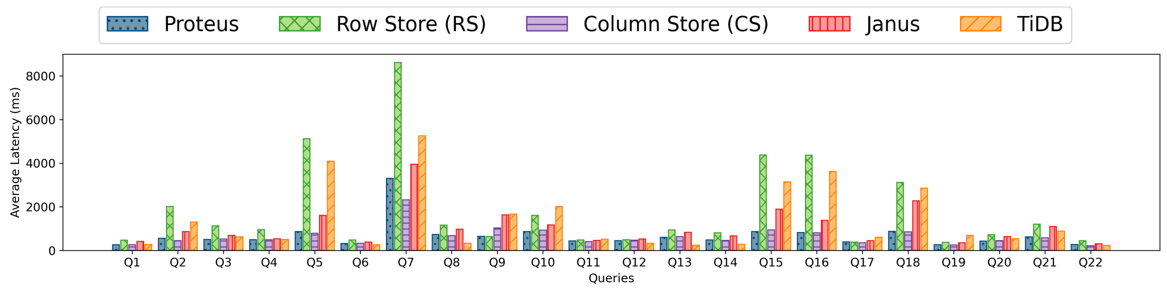


Figure 5.7: Average latency of each TPC-H query in the balanced CH-benCHmark (CH) workload.

storage is well-suited for hybrid workloads.

Observe that for the OLAP workload Proteus has similar query latency to CS, as shown in Figure 5.7 for most queries. Proteus has similar query latency to that of CS because these queries primarily execute over data stored in a columnar format. However, notable differences include Queries 2, 7, 17 and 22, where Proteus has query latency at least 20% greater than CS. Using Query 7 as an example, these queries features joins across multiple tables (six tables in addition to a self-join), multiple predicates (conditional on country names and delivery times), aggregation and ordering. By contrast, Query 9, which determines the profit made on items for each nation within a year, executes faster in Proteus and RS than CS. This query requires examining several attributes within the item, order and stock, so reconstructing these attributes from their columnar form increases the latency for CS. Despite these latency differences, Proteus remains competitive with CS on the overall OLAP workload while sustaining more than 2.2× its OLTP throughput, due to its adaptive storage techniques. Consequently, Proteus has superior performance on the hybrid CH workload.

Without a priori knowledge of the workload, Proteus makes effective adaptive storage layout decisions because its workload models capture how data is accessed and their access costs. To understand Proteus' performance, I discuss its storage choices.

First, Proteus heavily replicates read-only fact tables, such as the *nation*, *region*, *supplier* and *item* tables. This replication enables Proteus to execute many joins and aggregations locally, thereby reducing distributed data transfer. Proteus primarily stores these tables in a compressed column format in memory. Proteus places some of the partitions storing the *item* table on disk. The TPC-C workload has skew in the *items* ordered, and Proteus judiciously places infrequently ordered *items* on disk, relying on its zone-maps when performing joins to reduce disk accesses.

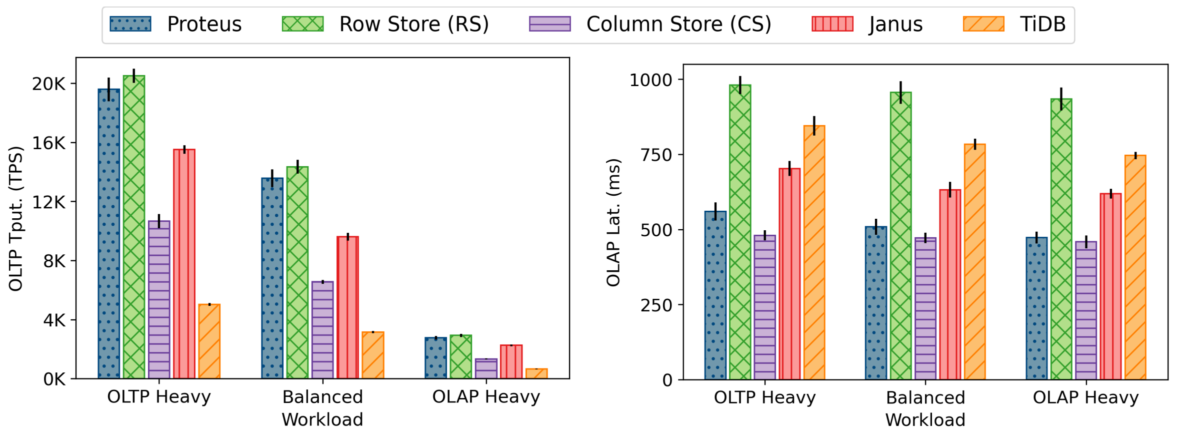
Second, as illustrated in the running example (Figure 5.2b), updates to the *orderline* table have a temporal relationship: recent *orderlines* are more likely to be updated. Tracking access frequencies over time allows Proteus to infer this relationship. The access patterns for the *orderline* and *order* tables are similar to the access pattern in YCSB: skewed OLTP operations access recently updated data with uniform OLAP queries over entire tables. Thus, Proteus makes suitable storage layout decisions — it employs row format for recent data, column format for read-mostly portions of the tables, and both row and column formats for partitions with relatively balanced accesses. Proteus leverages its decision reuse capability to repeatedly make decisions for partitions with similar access statistics. Proteus achieves OLAP latency similar to that of using only CS for OLAP queries featuring predicates that examine historical order information. Storing data in columnar format provides significant advantages for queries where scan costs dominate query latency (e.g., Query 6 that aggregates a column based on multi-column predicates). Proteus' storage layout adaptions allows it to execute OLAP queries over data primarily stored in a columnar format using similar efficient execution plans as CS but *without any* pre-configuration.

Third, Proteus leverages co-access likelihoods in terms of which data items are updated or joined together, e.g., *stock* and *orderlines* belonging to the same warehouse, to co-locate the storage of these partitions to the same site. This co-location of co-accessed data minimizes the overheads of distributed processing by allowing local joins of data[3] and single-site update transactions that avoid distributed commit. Unlike competitor systems that I advantaged with this information ahead of time to place data among sites, Proteus learns access patterns as the workload executes, making it robust to changes in these patterns (Chapter 5.4.3).

Fourth, Proteus maintains tables accessed mostly by OLTP transactions (e.g., *warehouse*, *district*, *history*) in a row-oriented format, adaptively partitioning and replicating data as necessary to mitigate contention and load effects. These storage layout decisions allow Proteus to execute OLTP transactions over data stored in a similar format to RS. By contrast, Janus' full data replication results in more data placed on the disk tier, which increases storage access costs, lowers OLTP throughput and raises OLAP query latency.

Proteus' significant performance improvements over the commercial TiDB system arises from these aforementioned factors. Specifically, Proteus' key differences with TiDB are: (i) TiDB has a static replication scheme and does not replicate read-only fact-tables to each database site. (ii) TiDB replicates all data items in both row and columnar storage

[3]Co-access frequencies are beneficial in equi-joins with foreign-key relationships, guaranteeing that if there is a local join match, then it is the sole match.



(a) Twitter OLTP Throughput (b) Twitter OLAP Latency

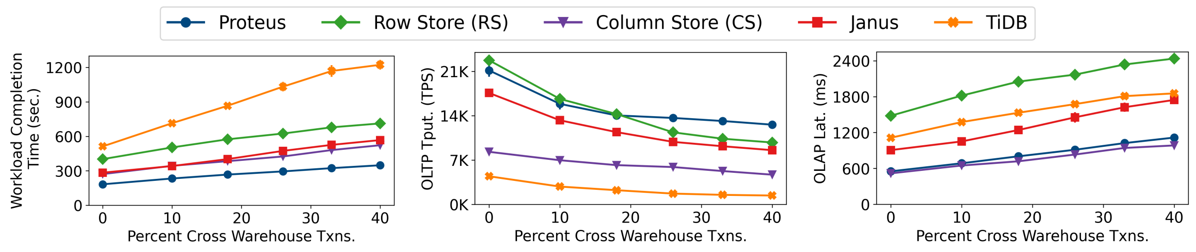
Figure 5.8: Twitter results showing OLTP throughput and OLAP latency for three HTAP mixes.

formats, needlessly applying updates to replicas of data that are not used in analytical queries. (iii) TiDB does not aim to co-locate co-accessed data and must incur distributed update transactions and distributed join processing, in contrast to Proteus' use of local equi-joins. Furthermore, TiDB does not support granular compression of columns, nor does it have the ability to operate over this compressed data directly.

Twitter

I evaluated Proteus using the Twitter benchmark and show average OLTP throughput and OLAP latency for all mixes in Figure 5.8. Proteus achieves throughput performance comparable with RS for all mixes (within 5% of RS OLTP throughput). Proteus has similar (within 7%) OLAP latency to CS in the balanced and OLAP heavy mixes. Remarkably, only Proteus achieves this high performance for *both aspects* of the hybrid workload.

Inserting new tweets dominates the OLTP workload, which results in significant contention on a small number of partitions. Consequently, I found that Proteus keeps recently inserted tweets in small partitions, in row format and on memory. Over time, Proteus merges these partitions into larger partitions and stores them as columns, as once inserted tweets become read-only. Consequently, OLTP transactions primarily execute over row format data while OLAP transactions execute over columnar data except for recent tweets. Moreover, Proteus rarely replicates data for this workload, so Proteus maintains all but



(a) Workload Completion Time (b) OLTP Throughput (c) OLAP Latency

Figure 5.9: Experiments that vary the percentage of cross warehouse transactions in the CH-benCHmark (CH).

the oldest tweets in memory, significantly improving performance over the fully replicated Janus.

The Twitter workload features a many-to-many relationship in its schema, making it difficult to partition the workload. The OLAP workload requires joining data in the presence of this many-to-many schema; for example, given a user u, get tweets from users that u is following. Data shuffling across nodes to perform joins reduces the relative effects of storage layout on OLAP latency. However, Proteus' ability to adapt data placement on nodes based on access patterns reduces the amount of data shuffling, allowing Proteus to remain competitive with CS in terms of OLAP latency while executing more than 2× as many OLTP transactions.

CH Cross Partition Transactions

Next, I study Proteus' sensitivity to the percentage of cross warehouse transactions in CH. Recall that in CH, clients are associated with warehouses, and *NewOrder* transactions vary the stock of ordered items, which are kept on a per-warehouse basis. By default, 10% of *NewOrder* transactions place orders to a different warehouse than the client's warehouse. Because of this locality, the best data placement scheme as determined by Schism [53] is co-locating data by warehouse. Consequently, as the number of cross warehouse transactions increases, OLTP transactions increasingly become distributed transactions. Moreover, increasing the percentage of cross warehouse transactions also increases the number of distributed joins, as several TPC-H queries (e.g. Query 7) join orders with stocks, which using Schism's data placement are not co-located at the same site.

Figure 5.9 shows the experimental results for CH as the percentage of cross-warehouse transactions is varied. Figure 5.9a shows the workload completion time as the percentage

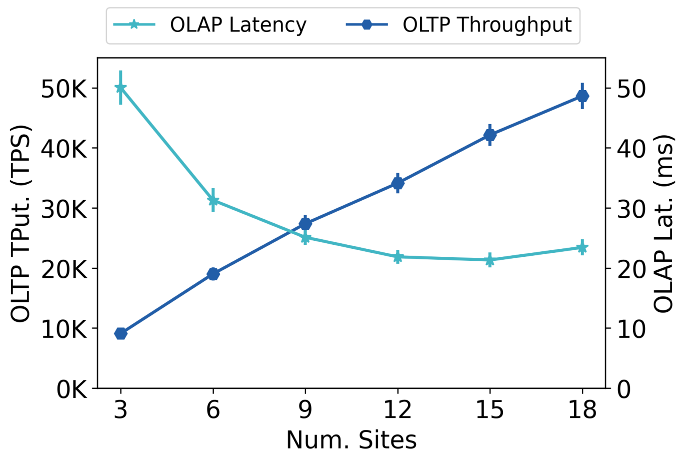


Figure 5.10: Proteus' Scalability

of cross-warehouse transactions increases. Observe that the workload completion time for all systems increase as the percentage of cross-warehouse transactions increases because the performance of both OLTP transactions (Figure 5.9b) and OLAP transactions (Figure 5.9c) decreases. However, Proteus' relative reduction in workload completion time to its next closest competitors increases from 1.45× to 1.63× as the percentage of cross warehouse transactions increases from 0% to 40%. Proteus achieves this relative increase in overall performance because Proteus adapts its storage layout, resulting in OLTP throughput outperforming RS by nearly 1.3× and achieving OLAP latency within 13% of CS. Specifically, Proteus increasingly replicates warehouse, district, customer and stock data among sites, which allows it to (i) reduce distributed join processing for OLAP queries, and (ii) dynamically change data mastership efficiently to reduce 2PC, allowing OLTP transactions to execute more efficiently. Proteus can perform this data replication because the decrease in OLTP throughput decreases the growth rate in the amount of stored data and because Proteus does not mandate data replication in two formats like Janus or TiDB.

Scalability

Figure 5.10 shows OLTP throughput and OLAP latency while scaling the number of data sites from 3 to 18 in step with the number of clients (30 per site) on the balanced YCSB workload. Proteus improves OLTP throughput by 5.3× as the number of data sites grows by 6×. Increasing the number of data sites reduces OLAP query latency by 2.2×, with the steepest fall in latency occurring when the number of sites grows from 3 to 9; with 9 sites,

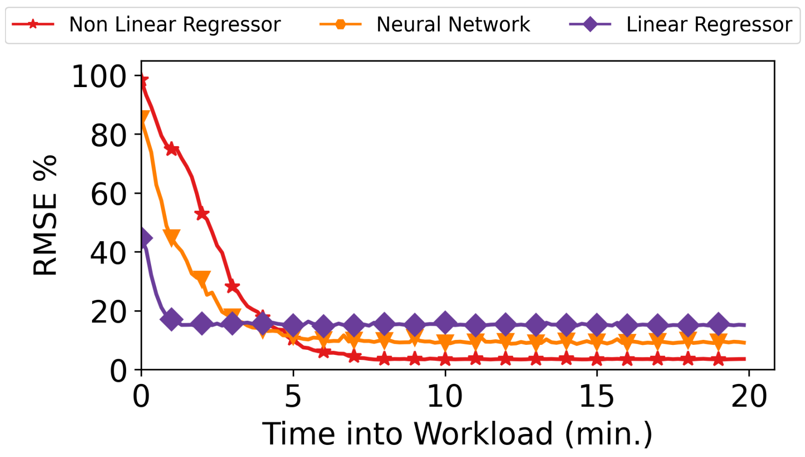


Figure 5.11: RMSE over time

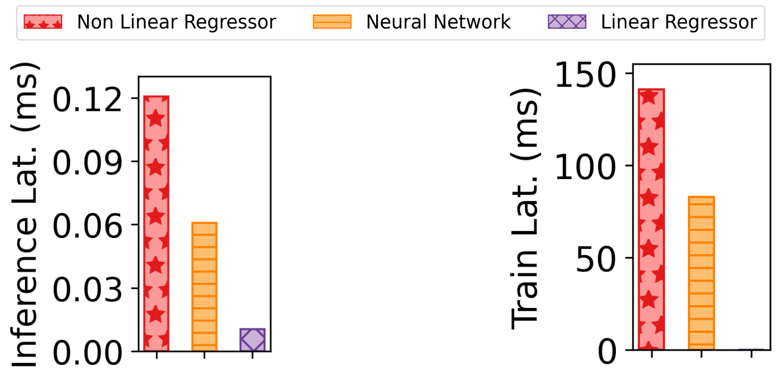
Proteus maintains most data in memory but at 3 sites, a majority of data resides on disk.

Cost Model Performance

I next evaluate the performance of the three different cost function estimators in Proteus: the linear regressor, neural network, and non-linear regressor. Figure 5.11 shows the root mean squared error (RMSE), $\sqrt{(y-o)^2}$, of each predictor normalized to each observation and averaged over each of the different cost functions throughout an experiment. Observe that by the end of the experiment, the linear regressor has the largest RMSE, and hence is the least accurate of the three models, while the non-linear regressor has the lowest RMSE and thus the most accurate of the models.

In Figure 5.11, all predictors converge to their average RMSE within 10 minutes. The non-linear regressor takes the longest to converge, while the linear regressor converges within one minute. These differences in model convergence rates are an important reason why Proteus uses multiple models when predicting costs. Proteus makes prudent decisions by considering predictions from multiple models, even if not all of the models have converged.

Although all three models have different degrees of accuracy, all three models have RMSEs that allow Proteus to distinguish between good and poor layout change decisions. If the cost predictions for two different decisions are similar, the actual cost is likely similar, but if one cost prediction is significantly higher than the other, it is likely to be a relatively poor decision.



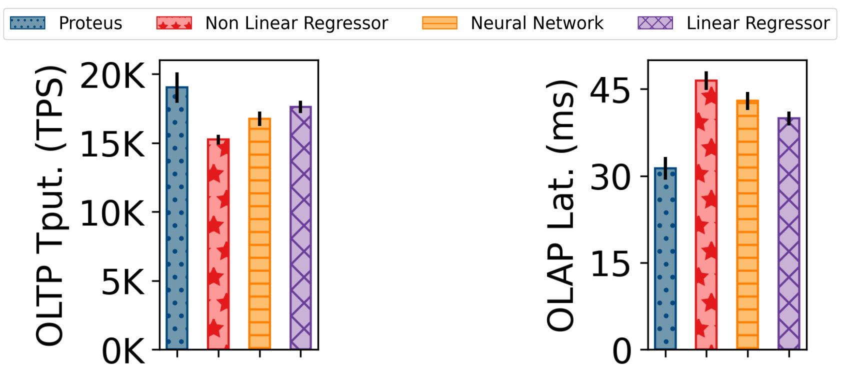
(a) Inference Latency (b) Training Latency

Figure 5.12: Latency of Proteus' Cost Functions

Increased model accuracy for the neural network and non-linear regressor, compared to the linear regressor, comes with a trade-off: increased latency to perform inference and training (Figures 5.12a and 5.12b). The linear regressor performs inference more than 10× faster than the non-linear regressor as it combines coefficients in a sum instead of invoking the kernel function. Moreover, the linear regressor performs a round of training more than 1500× and 2500× faster than the neural network and non-linear regressor, respectively. While incurring a slight decrease in model accuracy, these significant differences in latency justify using only the linear regressor when making predictions in a latency-sensitive situation such as selecting an execution plan. However, the more accurate neural networks and non-linear regressors are employed when generating storage layout change plans, which occur less frequently, and thus outside most requests' critical path.

The cost functions are also space efficient due to the small number of input parameters. The neural network has a larger model size (4 KB) from keeping a matrix of weights as opposed to the linear (64 bytes) and non-linear regressor (112 bytes) that keep a vector of weights.

To assess the effects of the cost function estimators in Proteus on system performance, I measured the OLTP throughput and OLAP latency for the YCSB workload. In these experiments, each decision requires invoking just one type of cost function estimators when compared to baseline Proteus that uses all three of Proteus' cost function estimators (Figures 5.13a and 5.13b). The linear regression predictor has the best performance of the three models in terms of OLTP throughput and OLAP latency. This result demonstrates the importance of inference time on overall system performance as the inference latency directly contributes to end-to-end request latency. Moreover, both the neural network and



(a) OLTP Throughput

(b) OLAP Latency

Figure 5.13: Performance metrics for Proteus' different cost predictors.

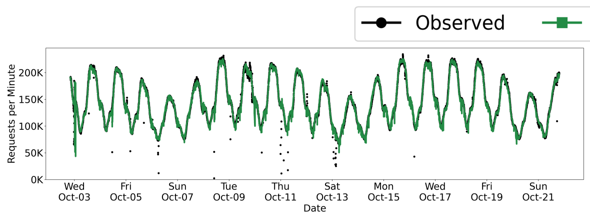
non-linear regression predictors require significantly more training data than the linear regressor to produce accurate cost estimates. Although the linear regressor produces less accurate estimates by the end of the experiment than the neural network and non-linear regressor, the linear regressor is initially more accurate. This initial accuracy allows Proteus to quickly begin adapting storage to the workload to deliver high performance. However, combining all three cost function estimators together yields the best performance.

Access Arrival Estimator Performance

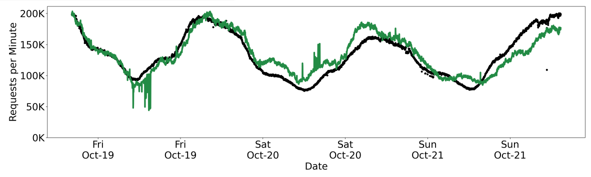
To assess the performance of Proteus' access arrival estimators, I used a real-world workload trace of Wikipedia accesses [195] and Azure [166]. For both SPAR and the hybrid-ensemble (HE) technique, I train on the same amount of data (two weeks for Wikipedia, one week for Azure) before predicting the number of requests that will occur each minute for the next hour. I then provide the models with the actual observed number of requests that occurred over the hour before predicting the next hour. Figures 5.14a and 5.14c show SPAR and HE's predicted number of requests compared to the observed number of requests (shown as dots).

For both workloads, SPAR (Figures 5.14a and 5.15a) and HE predictions (Figures 5.14c and 5.15b) match the observed number of requests. The Wikipedia workload features more regularity in the workload trend, and hence the RMSE (Figure 5.16a) is lower than for the more variable Azure workload (Figure 5.16b).

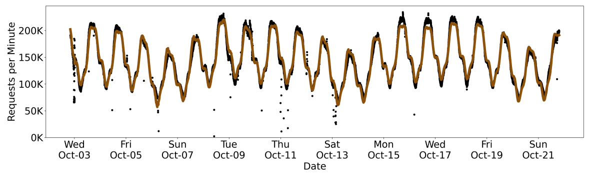
In both workloads, SPAR is more accurate than HE but requires prior knowledge of



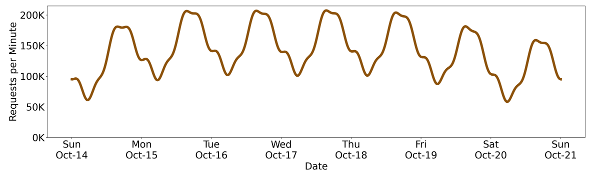
(a) SPAR Inferences



(b) SPAR Inferences with Wrong Period

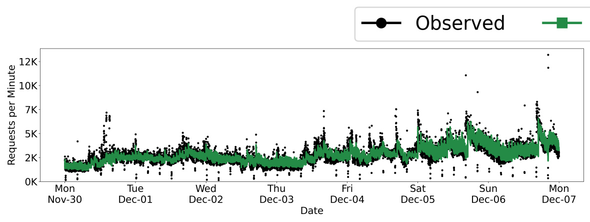


(c) Hybrid-Ensemble (HE) Inferences

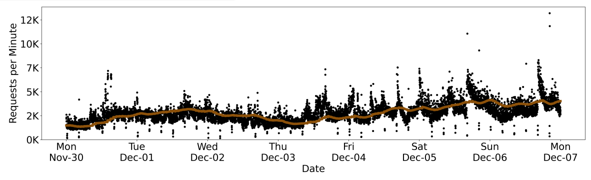


(d) RNN Fitted Pattern

Figure 5.14: Access Arrival Predictions for Wikipedia, using Proteus' access arrival estimators: SPAR and the Hybrid-Ensemble.



(a) SPAR Inferences

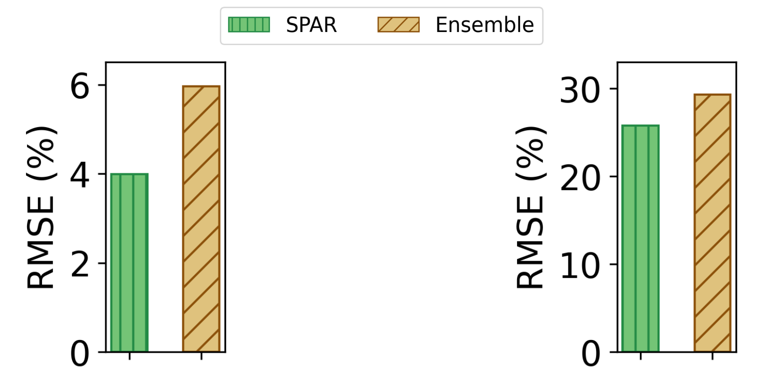


(b) Hybrid-Ensemble (HE) Inferences

Figure 5.15: Access Arrival Predictions for Azure, using Proteus' access arrival estimators: SPAR and the Hybrid-Ensemble.

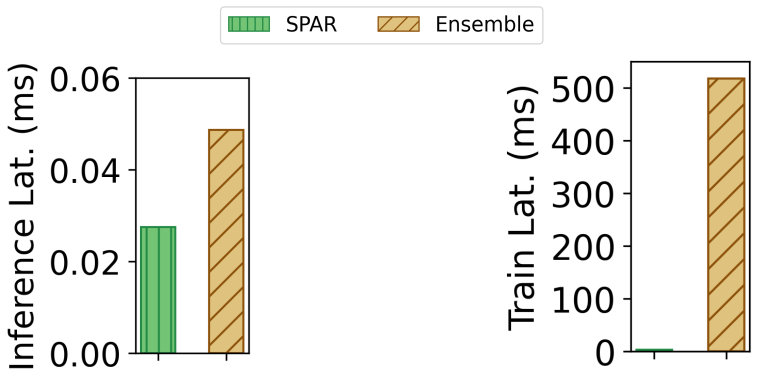
the workload, in this case, the periodic pattern follows a daily cycle. By contrast, the HE method learns the period from the data using the RNN. To highlight this difference in Figure 5.14b, I show SPAR's Wikipedia predictions over three days if configured to use a different period: the martian day (sol) that is 37 minutes longer than a day on earth. As shown, without proper prior workload knowledge, SPAR makes poor predictions, which results in an RMSE more than three times higher than for HE. These poor predictions occur because SPAR tries to align a periodic trend with the wrong period. By contrast, in HE, the RNN captures Wikipedia's periodic trend, as shown in Figure 5.14d over both the day and week without any user hints.

A key difference between SPAR and HE in the Azure workload is that SPAR is more



(a) Wikipedia Accuracy (b) Azure Accuracy

Figure 5.16: RMSE for Proteus' arrival estimators.

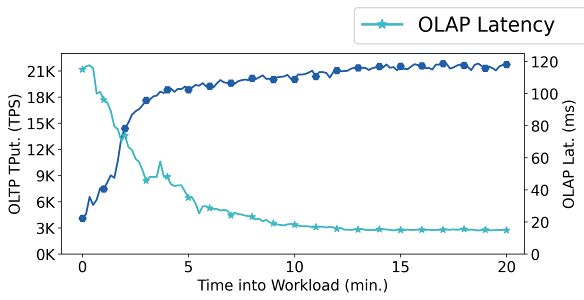


(a) Inference Lat. (b) Training Lat.

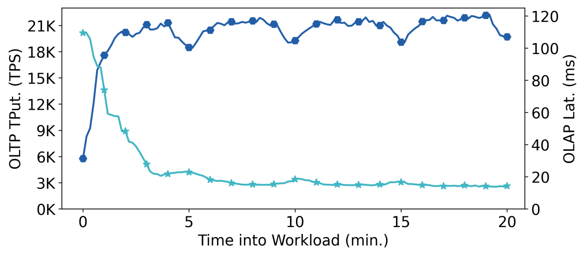
Figure 5.17: Latency for Proteus' arrival estimators.

variable in its predictions (Figure 5.15a) compared to HE (Figure 5.15b). This arises due to SPAR averaging specific prior observations based on the period, resulting in inherently noisy and variable predictions compared to HE's smoother predictions.

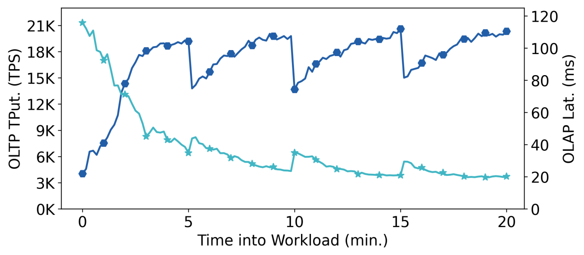
In Figures 5.17a and 5.17b I show the average latency of inference (60 predictions) and one round of training, respectively. SPAR is faster at both tasks; however, HE is competitive in inference latency, which is critical for making predictions. By contrast, training happens periodically and asynchronously, and the latency of a round of training (less than a second) is orders of magnitude smaller than the time it takes to gather the observations (minutes).



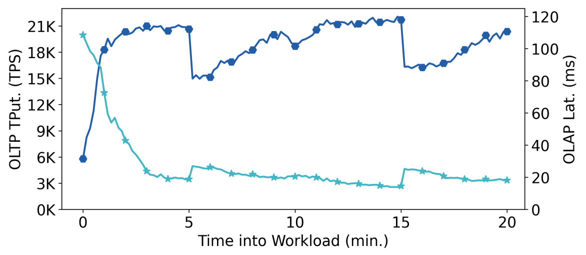
(a) Performance Over Time



(b) Performance with Shifting Hotspots



(c) Performance With No Access Arrival Estimators



(d) Performance with Wrong SPAR Period

Figure 5.18: Adaptivity experiments using shifting hotspot in a balanced YCSB workload.

Finally, the model sizes for the arrival estimators are space efficient with respect to the stored data. The SPAR predictor (3 KB) has a smaller memory footprint than the HE predictor (90 KB), as SPAR keeps a linear number of weights for predictions compared to HE's RNN which has multiple layers and stores a matrix of weights for each layer.

Adaptivity

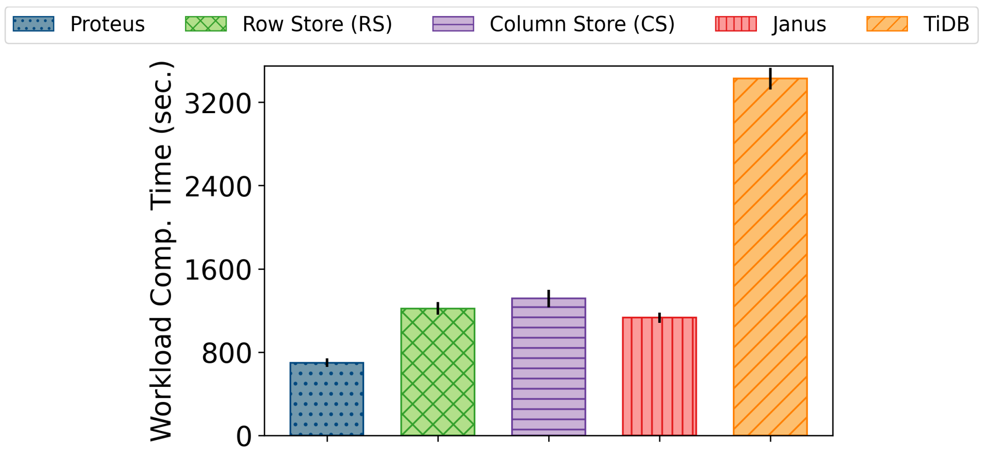
I studied Proteus' adaptive capabilities by examining its OLTP throughput and OLAP latency over time to understand its behaviour as it learns both the workload access pattern and cost model. Figure 5.18a shows Proteus' OLTP throughput and OLAP latency in the balanced YCSB workload. Proteus increases its OLTP throughput by 5.4× over the course of the workload while decreasing its OLAP latency by 7.9×. In this experiment, it takes Proteus roughly 3 minutes to reach within 15% of its peak OLTP throughput, and roughly 10 minutes to reach within 15% of its minimum OLAP latency. This difference is due to the skew in OLTP accesses compared to the uniform OLAP accesses; Proteus executes more layout changes for data primarily accessed by OLAP transactions. During this period, Proteus rapidly builds both its workload model to understand data access patterns and its cost model to estimate operation latencies. Even on a cold start, Proteus' cost model is accurate and averages a root mean squared error (RMSE) of 11% of the observed average latency, allowing Proteus to distinguish between good and poor layout change decisions.

Figure 5.18b repeats the experiment from Figure 5.18a with three changes: (i) the centre of the OLTP skew shifts every 5 minutes following an hourly cycle (ii) Proteus' data access latency models are initialized using the end model state resulting from the experiment in Figure 5.18a (iii) Proteus' access arrival estimate model is pre-trained using the historical access pattern of the workload. Compared to Figure 5.18a, Proteus reaches within 15% of its peak OLTP throughput in just 1 minute, and within 15% of its minimal OLAP latency in 6 minutes (Figure 5.18b). Slight shifts in performance are visible both before and after the 5, 10, 15 and 20-minute marks due to the workload shifts occurring at these same time points. Proteus begins executing storage layout changes predictively in anticipation of the workload shift due to high confidence in changes to the workload access pattern. The small performance shifts arise primarily due to (i) storage layout changes consuming resources and (ii) predictive storage layout changes that amortize costs over time to provide beneficial layouts for the future.

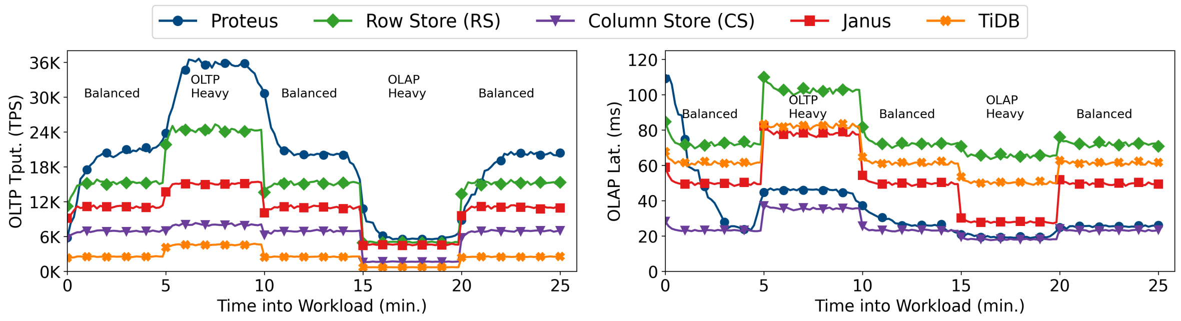
By contrast, in Figure 5.18c, I disabled Proteus' ability to predict access arrival estimates and can merely react to the workload changing. Finally, in Figure 5.18d, I configured Proteus using SPAR but with a ten-minute period, rather than the 5 minute period that aligns with the workload shift.

In contrast to Figure 5.18b, which has an accurate access arrival estimator, both Figures 5.18c and 5.18d, suffer from degradations in performance when the workload shifts. Specifically, there is a 35% decrease in OLTP throughput and a 1.47× increase in OLAP latency when the workload shifts. By contrast, in Figure 5.18b OLTP throughput degrades by just 15% and OLAP latency increases by just 1.21×. This experiment demonstrates the benefit of predicting access arrival times and predictive storage layout changes on overall system performance, allowing greater amortization of the costs of performing storage layout changes.

Observe that Proteus without predictions (Figure 5.18c) responds to the workload shifts at the 5 and 15-minute mark faster than Proteus with a misconfigured workload period (Figure 5.18d). With the misconfigured period, Proteus delays responding to the workload shift with storage layout changes because of the disagreement in predicted access patterns (intentionally incorrect) and the actual access patterns results. However, in time, Proteus adjusts its storage layout, and performance improves. The adverse effects of misconfiguration of the workload period using SPAR highlight the benefit of Proteus' hybrid-ensemble



(a) Workload Completion Time



(b) OLTP Throughput Over Time (c) OLAP Latency Over Time

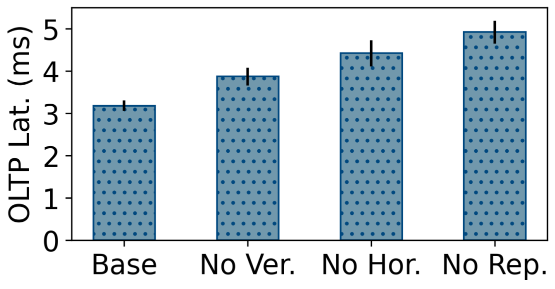
Figure 5.19: Shifting workload mix experiments using YCSB — mix shifts every 5 mins over the course of the experiment.

predictor that learns the workload period.

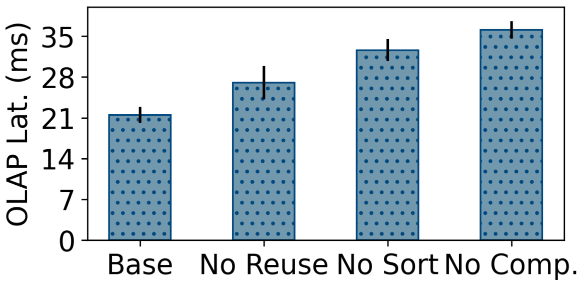
In Figure 5.19, I examined Proteus' ability to predict and respond to shifts in the workload mix over time. In this experiment, I followed the same methodology used for the experiments in Figure 5.18b but shift the workload mix every 2,000 OLAP transactions[4] in Figure 5.19a and every five minutes in Figures 5.19b and 5.19c. I measured workload completion time, OLTP throughput and OLAP latency over time.

Observe that Proteus completes this workload faster than all of its competitors, including 1.6× faster than Janus, which does not adapt to the workload but keeps copies of

[4]2,000 OLAP transactions are used since the same number of OLAP transactions execute over the five shifts as in the other experiments.



(a) OLTP Ablation



(b) OLAP Ablation

Figure 5.20: Proteus system latency under an ablation study using YCSB.

all data in column and row form. Examining performance over time, observe that as in Figure 5.18b, Proteus rapidly improves both OLTP and OLAP performance as it adapts to the workload. A key difference between Proteus and its competitors is how Proteus behaves before the workload shift occurs: Proteus predictively and autonomously begins to change storage formats in anticipation of the workload change. For example, when shifting from the balanced to OLTP heavy mix, both Proteus' OLAP latency and OLTP throughput increase. These performance changes occur as Proteus predictively executes layout changes from columnar to row format data.

Ablation Study

Figures 5.20a and 5.20b show an ablation study on Proteus' ability to make adaptive storage decisions by independently and categorically removing different techniques while using the YCSB workload. Shedding Proteus' ability to vertically and horizontally partition increases OLTP latency by 1.2× and 1.4×, respectively, as these techniques help mitigate contention effects within, and across, rows. Removing Proteus' ability to add or remove replicas also affects its OLTP latency. Proteus leverages replicas for two purposes: (i) replicating frequently updated data among sites to distribute load, and (ii) replicating data partitions with roughly equal OLTP and OLAP access frequency in both row and column format to provide storage formats for both workloads.

Figure 5.20b shows the effects of the ablation study on OLAP latency. Removing compression in Proteus increases OLAP latency by 1.7×: less data is kept in memory and cannot be operated on in compressed form. Proteus often stores columnar data using per column sort-orders as the OLAP workload features inequality predicates; removing this increases OLAP latency by 1.5×. Proteus' decision reuse benefit is also shown: applying

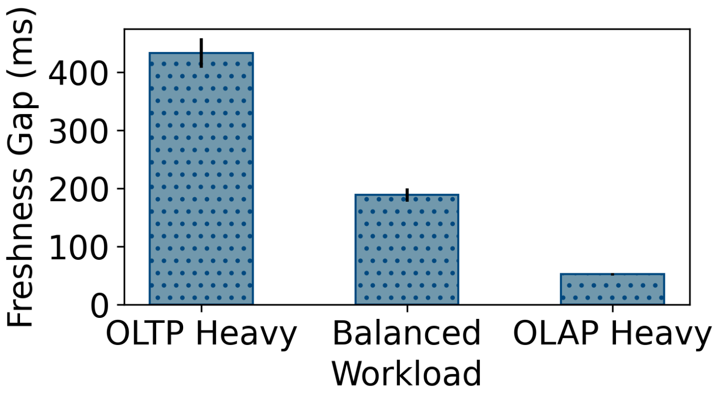


Figure 5.21: Proteus' freshness gap when performing OLAP scans.

execution plans and layout changes to partitions with similar access statistics reduces latency.

OLAP Freshness Gaps

Proteus targets real-time analytical processing, where transactional updates are immediately available to analytical queries. To measure the effectiveness of Proteus at providing real-time analytical processing, I measured the average freshness gap of OLAP queries. Specifically, I modified the YCSB benchmark so that: (i) OLTP transactions set every updated value to a timestamp, (ii) OLAP queries return the smallest value read in the scan (i.e., oldest timestamp observed). I recorded the values set by OLTP transactions and values returned by OLAP transactions, along with the (real) time that the OLTP transaction committed and the OLAP transaction began. After the experiment, I combined this observed state and recorded the difference between (i) the smallest value read in each OLAP query (oldest timestamp observed) and (ii) the most recent commit time before the beginning of the OLAP transaction that updated values in the range of the OLAP scan. Hence, if the OLAP scan reads the freshest data available, I recorded 0 and otherwise record a value that indicates how stale the OLAP scan was. The average of these values represents Proteus' *freshness gap*.

Figure 5.21 presents the freshness gap for Proteus over the three YCSB workloads. Observe that in the balanced YCSB workload, Proteus' freshness gap is less than 200 ms. In the OLTP heavy and OLAP heavy workloads, the freshness gap is approximately 450ms and 50ms, respectively. Hence, OLAP queries in Proteus observe a fresh state, satisfying

Operation	**Proportion of Time Spent**	**Avg. Latency (ms)**	**Frequency (per 1000)**
OLTP Transaction	47.1%	6.37 ± 0.1	991 ± 24
OLAP Transaction	48.1%	685 ± 40	9.4 ± 0.6
Storage Format Change	2.14%	14.0 ± 1.1	20.4 ± 0.8
Storage Tier Change	0.51%	12.9 ± 0.8	5.3 ± 0.2
Sort or Comp. Change	0.04%	20.7 ± 1.6	0.26 ± 0.01
Partition Change	0.07%	4.56 ± 0.4	2.2 ± 0.12
Replication Change	1.96%	36.9 ± 2.9	7.1 ± 0.6

Table 5.5: The proportion of time spent, average latency and frequency per operation for the CH-Benchmark workload.

the real-time analytical processing requirement. Proteus' low freshness gap is primarily due to (i) efficient update propagation and (ii) performing OLTP transactions directly on column data if OLAP queries primarily access the data. Proteus has a higher freshness gap for the OLTP heavy workload because it propagates and applies more updates to replicas.

Storage Layout Change Operation Overhead

Table 5.5 summarizes the proportion of time spent on transactions and storage layout change operations in the balanced CH-benCHmark workload. Proteus balances the proportion of time spent on executing OLTP and OLAP transactions, demonstrating the benefit of using OLTP and OLAP-specific thread-pools to execute requests. These storage layout changes are efficient as they execute about as quickly as an OLTP transaction. The most frequent changes involve changing formats, tiers, and partition replicas.

In Table 5.6, I summarize the proportion of time spent planning transaction execution, layout change plans, and executing layout change plans. Proteus' decision reuse allows efficient selection of physical execution plans for transactions, which take about 1% of overall system time. For fewer than 10% of transactions, Proteus generates a layout change plan. Layout change plans in response to OLAP transactions take longer to develop than OLTP transactions due to the number of data items accessed in OLAP transactions. Finally, about 1% of transactions execute layout change plans, which execute in 30 ms and 235 ms on average for OLTP and OLAP transactions, respectively. These plans execute quickly due to the low latency of individual layout change operators. Together, Proteus spends less than 5% of the time planning and performing storage layout change plans as Proteus amortizes the costs of layout changes across transactions.

Operation	**Proportion of Time Spent**	**Avg. Latency (ms)**	**Frequency (per 1000)**
OLTP Physical Execution Plan Generation	1.32%	0.18 ± 0.01	991 ± 24
OLAP Physical Execution Plan Generation	0.88%	12.7 ± 1.1	9.4 ± 0.6
OLTP Layout Change Plan Generation	1.02%	1.62 ± 0.8	84.9 ± 5.7
OLAP Layout Change Plan Generation	1.14%	56.8 ± 4.2	2.7 ± 0.34
OLTP Layout Change Execution	3.01%	30.8 ± 3.7	13.1 ± 0.96
OLAP Layout Change Execution	1.19%	235 ± 27	0.68 ± 0.03

Table 5.6: The proportion of time spent, average latency and frequency of planning and executing layout changes for the CH-Benchmark workload.

5.5 Summary

In this Chapter, I presented Proteus, a distributed HTAP database system that adapts data storage layouts to deliver excellent performance for hybrid workloads. Proteus adaptively decides on row- or column-storage formats, storage tier, and whether to employ optimizations such as sorting or compression in addition to data replication and partitioning schemes. Proteus makes these decisions on-the-fly using learned workload models that predict access costs and when data will be accessed. Using this information, Proteus makes cost-driven decisions about adapting its storage. Proteus reduces HTAP workload completion time by up to 70% over prior approaches while precluding the use of static storage layouts. Proteus' ability to generate and adapt storage layouts dynamically without prior workload knowledge demonstrates its self-managing ability.

Chapter 6

Related Work

Building adaptive distributed DBMSs blends together techniques from the database systems, distributed systems, and machine learning communities. I now examine related work across these areas.

6.1 Distributed Database Management Systems

The original data management systems, such as IBM IMS were designed to be used on a central disk-based system [145]. However, the distributed SDD-1 system [202, 95, 145] demonstrated the benefit of distributing data across nodes in a cluster. SDD-1 split data into partitions using both horizontal and vertical partitioning on disk and used a transaction coordinator to provide transactional serializability using conflict detection. Systems such as R* [199] and distributed INGRES [185] further advanced the work of SDD-1 and introduced distributed coordination protocols such as distributed two-phase locking, reading from snapshot state, and distributed query optimization [145].

The advent of faster networks and in-memory storage prompted the redesign of distributed database systems and resulted in systems such as H-Store [96], and Amazon Aurora [196], the partitioned and replicated DBMSs described in Chapter 1.

6.2 Distributed Coordination

I first examine techniques and systems that aim to minimize the overheads of distributed transaction processing.

To mitigate the blocking effect of 2PC, some systems [94, 73, 161] speculatively execute transactions. Specifically, these systems make the updates made by transaction T visible to subsequent transactions T' before receiving the global commit of T. However, speculative transactions are also blocked from committing until the global commit of T arrives.

Coordination avoidance [28] exploits application invariants [51] and commutative data types [179, 111] to avoid distributed transactions. These invariants and data types allow sites to merge diverging updates asynchronously, without coordination independently. For example, a record tracking the total number of sales can safely merge updates without coordination, provided that there are no queries that require the precise number of sales at a specific point in time. Unfortunately, not all applications have these invariants, so distributed transactions still arise.

Many distributed DBMSs replicate data for fault tolerance, and these replication schemes require a separate protocol to ensure correctness, such as Raft or Paxos. TAPIR [211], and Janus [133] address the overheads of distributed transaction processing by coupling the transaction consistency protocol with the data replication protocol that is necessary for fault tolerance. However, these systems do not guarantee single site transaction execution, statically assign master copies of data to nodes and do not support mastership changes.

Deterministic databases [194, 162] eliminate distributed communication by grouping all transactions submitted to the system within an epoch and then executing them as a batch, which increases transaction latency. Deterministic databases also support limited transactional operations [163]. To avoid synchronization costs among machines in a deterministic database, T-Part [204] partitions and re-orders transactions in batches, so that writes arrive at remote sites earlier. STAR [119] replicates data into both a single-master and partitioned multi-master format, then groups transactions into batches and divides them into either the single-master or partitioned multi-master execution batch. Such batch-based systems do not execute transactions immediately when they arrive, thereby increasing transactional latency.

Repartitioning systems [53, 62, 174, 191, 171, 24, 40, 134] periodically change the location of data items or disk blocks to minimize the number of transactions that cross site boundaries as the workload changes. However, distributed transactions are still required unless the workload is perfectly partitionable. Dynamic repartitioning is fully discussed in Chapter 6.4.

In a bid to decouple transaction processing from data storage, shared-data architectures such as Tell have been proposed [116] in which transactions forward reads and writes to a shared-storage subsystem supported by a fast network. This approach requires the shared-storage system to support atomicity and consistency guarantees for transactional data accesses. Despite presenting a unified storage interface, data access to remote storage nodes can incur network communication overhead. G-Store [54] provides atomicity and consistency for multi-key operations on databases that use shared storage via a 2PL-like protocol. LEAP [114] extends the application of these transactional properties beyond the lifespan of a single transaction, which reduces the overhead of future transaction execution at a node.

Some systems utilize caches to execute transactions locally without distributed coordination. NuoDB [2] executes transaction updates on locally cached copies of data partitions and propagates any updates to any remote caches. To ensure transactions do not conflict, NuoDB requires transactions to communicate with the (possibly remote) leaders of each updated partition in the transaction. In NuoDB, the location of a partition leader changes only if the site removes the partition or shuts down; hence distributed coordination may be performed for many transactions. Sundial [208] introduces data caches into a partitioned system to minimize the latency of distributed transactions. Sundial uses *logical leases* as the basis of both the cache coherence protocol and its optimistic concurrency control scheme. Similarly, MaaT [124] uses logical timestamps to change the commit order among transactions through explicit coordination. However, these systems still incur distributed commit latencies unless the workload is perfectly partitionable.

Several systems exploit advances in hardware such as low-latency remote memory accesses [210, 60, 44, 198], programmable network switches [112], hardware transactional memory [44], or non-volatile memory [60, 198] to improve throughput in distributed databases. Such hardware allows efficient remote data access but still requires expensive distributed protocols to coordinate and acquire locks, such as distributed 2PL through RDMA operations. This thesis focuses on techniques that do not rely on specialized hardware technologies.

6.3 Data Replication

I now examine systems that adjust which data is replicated, and where.

The adaptive data replication (ADR) algorithm [201] is early work in the area of dynamic replication. ADR changes the replication scheme of data items dynamically, based on the read-write pattern of the data item in an online fashion. The ADR algorithm is decentralized, so each node in the system makes replication decisions locally based on local statistics. As ADR is a decentralized system, each node responds to client requests by either returning the data if it is local or asking its neighbours if they have a copy of the data. Each node decides to add a replica if it finds that over time, the number of read requests that it cannot satisfy locally outnumbers the number of write requests to the data. The ADR algorithm is used in non-transactional storage systems [82, 131], and key-value stores [139]. Moreover, ADR does not make replication decisions that are integrated with master placement and data partitioning decisions.

NashDB [128] adaptively replicates read-only data for OLAP workloads periodically using an economic model that aims to balance replica supply to workload demand based on user-submitted query priority. Notably, NashDB does not support updates. In NashDB, each query has a price value (priority), and each record necessary to respond to a query is allocated a fraction of the query's value. Hence, given every query, a given record has a corresponding value to the workload. NashDB then replicates each record proportionally to its value; hence, records accessed frequently or by higher priority queries are more frequently replicated and thus more easily accessible.

Rabl and Jacobsen [156] propose an offline algorithm to select replicas in a distributed DBMS based on the potential theoretical speed-up a replica provides for parallel query processing. A query must be processed at a single node in their model and requires all data at that node. Hence, if two nodes store the data required for the query, twice as many queries can execute compared to if only one node stores the required data. Rabl and Jacobsen propose an analytical cost model that quantifies data replication's potential speed-ups and uses this model to develop an automatic replica selection algorithm.

At a global scale, distributed systems must place replicas of data worldwide so that both access and update latencies are minimized. Data placement is a challenging problem as clients can be spread globally, and as more copies of data are created, the cost of keeping data consistent in the face of updates increases. Sharov and colleagues [176] propose an offline optimization formulation for leader replica placement. The GPlacer offline tool [209] also addresses replica placement but generalizes beyond leader placement to other non-leader replica protocols such as Paxos. These tools use the K-means heuristic to place replicas such that the average latency to access a replica is minimized. However, as these tools make offline decisions, they cannot respond to changes in data access over time.

Data caching is a particular form of adaptive replication, in the sense that what is replicated, or cached, can adapt based on access history [208, 124, 2]. To ensure consistency, systems that cache data must either update or invalidate the caches. Although some systems such as Sundial [208] integrate the consistency mechanism into their concurrency control protocol, most caching solutions decouple cache invalidation from transaction execution. Moreover, these systems fail to make integrated replication decisions and continue to suffer from distributed coordination overheads. Predictive caching [148, 37, 66] aims to further improve the performance of cache-based systems by placing data items that will be accessed in the cache ahead of time.

Finally, data replication is often used to ensure high-availability and fault tolerance. Slogger [22] is a system that supports linearizable, and hence prefix consistent, continuous backups in a static geo-replicated environment. Critically, Slogger aims to maintain a small lag on the replica site to minimize data loss in a disaster. Slogger achieves these goals by using synchronized timestamps in the write-ahead log and watermarks to ensure the replica applies updates from the primary in the correct order. In distributed storage systems, erasure-coding is often used as a replacement for replication because it can provide lower storage overhead [83, 188]. EC-Store [11] is an adaptive erasure-coded storage system that dynamically places and moves data fragments to co-locate co-accessed data to minimize straggler effects and reduce access latencies. EC-Store does not support database workloads, as it focuses on non-transactional block storage operations.

6.4 Data Partitioning

I now examine systems that group data items together into partitions as a logical data storage unit.

There are several offline tools for forming data partitions, including Schism [53] and Sword [155]. These tools receive an offline trace of transaction accesses to data to form an access graph. Nodes in the graph represent data items, while weighted edges represent the number of co-accesses between data items. Given this access graph, the tools use graph partitioning algorithms, such as Metis [98] that aim to minimize edge cuts or cross partition data access. The resulting partitioned graph represents the set of tuples that should be grouped together. Schism [53] further simplifies the resulting data partitioning by creating a decision tree based on the mappings of tuples into a set of range partitions. As these tools produce a static data partitioning scheme, the tools cannot respond to changes in workloads such as shifting hotspots.

Several data partitioning systems [171, 190, 62] are designed for the shared-nothing H-Store system. In H-Store, each CPU core executes a transaction serially, and the transaction accesses data partitions assigned to the CPU core. Hence, data partitioning significantly affects system performance [94, 76]. Clay continuously forms partitions using a partitioning algorithm that groups data together into clumps and migrates the clumps across sites to balance the load. Clay forms clumps in three phases: (i) identifying frequently accessed tuples, and then (ii) identifying tuples frequently co-accessed with tuples in a clump, and (iii) repeating the second stage until the clump cannot be expanded further or reaches a maximum size. By grouping together tuples frequently co-accessed, Clay aims to keep transactions single-sited. However, as these systems are designed for H-Store, they require two-phase commit for multi-partition transactions and do not support replication for updated tables.

Object-oriented databases use semantic locking [135, 164] to improve the concurrency of transactions. However, such systems require significant information provided by a user, including the hierarchical relationship among data items (classes) and the semantics of operations on these data items. Hence, semantically and hierarchically related data items are grouped together and share resources such as logical locks. Using this knowledge, the semantic locking protocol acquires these locks hierarchically based on data item semantics. Concurrency among operations can also be increased by leveraging semantic knowledge of operations through user-defined compatibility matrices or prior knowledge of the commutativity of operations. Efficiently executing transactions requires semantic locking systems to provide additional concurrency control on specially defined conflict and commutativity tables.

Bigtable [41] is a distributed storage system for structured tabular data. Bigtable supports dynamic partitioning of this tabular data to ensure partitions have similar sizes. Moreover, Bigtable supports elastically adding or removing database sites, which triggers re-distribution of data partitions to distribute load. However, Bigtable does not support multi-row transactional updates nor dynamic replication.

While the prior discussion focuses on horizontal data partitioning, that is, on the individual data item level, data can also be partitioned vertically by data item attributes.

In STOv2 [87] the authors note that many rows have attributes that are infrequently updated. However, row-based concurrency control protocols produce conflicts between transactions that read these rarely changing attributes and transactions that update other attributes in the row. Row-splitting vertically partitions rows to eliminate these conflicts. STOv2 implements offline rows splitting in timestamp-based concurrency control schemes by assigning separate timestamps to infrequently updated attributes and other attributes in a row.

Database cracking [89, 169, 170, 74, 91] automatically vertically partitions databases based on the presence of predicate queries in an online environment. Database cracking highlights a consequence of data partitioning: changes need to be made to data access operators, for example, pushing down *cracked* database operators that operate on vertically partitioned data.

In distributed join processing, if the data to be joined are not located on the same node, then the DBMS must shuffle data among nodes so that the join can be processed. By contrast, if the data partitioning scheme matches the join predicate, the data to be joined resides on the same node, then the join can be processed locally. Several systems [80, 59, 118], including Amoeba [173] and the QD-Tree [207], adapt data partitioning in response to the read-only workload to minimize the amount of data shuffling necessary to execute joins. Amoeba and the QD-Tree work by hyper-partitioning data along many attributes, such that each record within a partition satisfies the same predicate. When possible, the systems then store partitions that satisfy the same predicates used in joins at the same site. Otherwise, the system tracks the predicates associated with the partition to quickly identify the data that must be shuffled to perform the join.

Jigsaw [97] uses prior workload knowledge to statically partition data both horizontally and vertically using recursive partitioning to form irregular data partitions. Jigsaw uses these partitions to execute read-only OLAP queries on a single node. Jigsaw uses the irregular partitions to execute queries partition at a time, a hybrid approach to query processing between tuple-at-a-time and vectorized processing [100]. Jigsaw does not support updates, does not replicate data and does not consider multi-table accesses.

6.5 Adaptive Storage for HTAP

I now discuss systems that change how data is stored in a DBMS, focusing on hybrid workloads.

HYRISE [71], and the flexible storage manager (FSM) [26] are single node DBMSs that support hybrid workloads by storing data in either row or column-oriented format on a single node. HYRISE stores data in variable-width columns based on access frequency, while FSM stores recently written data items as rows and read-only data as columns. FSM stores data in physical tiles and uses logical tile algebra to support query processing over its data layout. However, neither HYRISE nor FSM replicate data and thus, each data item is stored in precisely one format. Consequently, these systems suffer if a data item is accessed by both OLTP and OLAP transactions. Moreover, as single-node DBMSs, these systems do not support workloads where the data size is larger than the capacity of a single node.

L-Store [168] is a single node DBMS that stores data primarily in columns but uses lineage information to support efficient updates to the stored data. L-Store strictly keeps data in one copy and format of data and thus executes queries without requiring layout-aware information. The tracked lineage information allows L-Store to efficiently identify the latest version of each record, allowing the contention-free merging of historical data with recent data.

Similarly, Casper [27] stores data in columnar format for hybrid workloads on a single node DBMS. Based on collected workload history, Casper changes storage columnar formats by changing partition sizes, ranges, sort order, and update policies. However, Casper does not support storing data in both row and column format, either selectively or simultaneously. Hence, Casper's performance suffers if a data item is accessed by both OLTP and OLAP transactions or only OLTP transactions.

H_2O [21], and OctopusDB [92] propose storing data in both row and column formats on a single node based on the workload. However, H_2O considers only read-only workloads and uses view materialization to store data in different formats to optimize the execution of scans and joins. Consequently, H_2O does not support hybrid workloads that contain OLTP transactions. OctopusDB is a simulation tool and relies on user hints of the workload to select the storage format of stored data.

Fractured mirrors [158] identified that DBMSs could maintain a complete replica of data in a different storage format, such as one copy in row-storage and one copy in column-storage. Thus, one copy would benefit OLTP workloads while the other would be beneficial for OLAP workloads. Fractured mirrors proposed this data replication in the context of systems that used RAID mirrored storage. Janus [25] similarly fully replicates data but proposes a graph-based dependency management algorithm to ensure consistent merging of updates. Other fully replicated distributed DBMSs include F1 [206], BatchDB [125], and TiDB [84]. While most replicated systems execute the OLTP workload on the row-storage copy of data and the OLAP workload on the columnar copy of data, TiDB uses a cost model to decide where to execute a given request. Crucially, all these systems make static data placement and storage decisions.

Umbra [138] is a single-node DBMS that gracefully handles storage of data beyond the capacity of memory by using a low-overhead buffer manager for SSD and disk-based storage. Umbra's buffer manager stores data using variable-sized pages, which simplifies the rest of the DBMS engine as it can operate as though data is stored consecutively in memory. Umbra relies on operating system primitives to provide contiguous virtual memory addresses, although data may be fragmented in physical memory space. Importantly, Umbra does not support adapting storage format based on the workload, as its focus is changing the storage tier where data is stored. Mosaic [197] is a storage engine for Umbra optimized for OLAP-dominated workloads. Mosaic makes user-prompted static storage decisions of where to place columnar data in attached storage devices. Mosaic requires a trace of the workload to make Pareto-optimal placement decisions with respect to I/O throughput and the monetary cost of storage. Hence, Mosaic is an offline tool that does not respond to workload changes. Moreover, Mosaic supports only columnar data storage and does not support adapting between storage formats.

6.6 Other Forms of Adaption in DBMSs

Finally, I discuss other forms of adaptation that are present in DBMSs. While these adaptations have not been the focus of this thesis, they provide further examples of the benefits of adaptive DBMSs and adaptation techniques. DBMSs utilize predictive components and models to understand the workload. The key commonality in these systems is that they perform a cost-driven “what-if” analysis to quantify the effect of the proposed action on system performance [42].

Several systems [190, 203, 191, 171] propose mechanisms for automatic distributed DBMS elasticity in the cloud. That is, deciding when to add or remove database nodes based on the workload so that the system uses only the resources necessary to satisfy the workload demands. Unlike most elastic systems that react to changes in load, P-Store [190] predicts the upcoming load and predictively adds or removes nodes based on these predictions.

Indices are used to accelerate query processing in DBMSs. Several tools [55, 140, 91, 42, 18] exist to identify which indices should be used and to automatically generate, deploy, or tune these indices. Similar tools exist to select the set of views to materialize based on the workload [72]. A key challenge faced by these tools is detecting and correcting performance regressions caused by poor choices.

DBMSs use cost models to generate query execution plans, and recent work has examined learning these cost models. Initial work to use learning aimed to replace individual cost functions inside the DBMS with learned replacements [180]. Black-box approaches aim to replace the entire query optimizer or cost models with learned components [127, 129, 126]. Deep reinforcement learning-based approaches in this domain [130, 142, 104] consume a plan tree structure as input, including join predicate information, while relying on the hidden layers to capture and learn the relevant information. However, the complexity of the model vastly increases training time, precluding its use in an online environment [130].

Administrators configure DBMSs using configuration knobs, for example, selecting how much memory to use as a buffer. Recent work examines the automatic tuning and adjustment of these knobs using machine learning algorithms [20, 123, 61]. These tools face the challenges of facing dependencies among decisions and delayed effects of changing the knobs. Consequently, it is difficult to attribute the effects of performance to any individual knob. These challenges are similar to the ones faced when attempting to adapt how and where data is stored.

The Sentinel [65] and Dendrite [67] systems provide a mechanism for administrators to bolt-on adaptivity to non-adaptive database systems. These systems operate by recording database and system events (e.g., page writes or CPU utilization) and triggering administrator-defined rules when the behaviour of the recorded event matches the specified rules. For example, an administrator may define a rule that changes where queries execute when CPU utilization exceeds a specific threshold. Although this approach allows for adaptivity without changes to core database internals, it leaves it to the administrators to pre-define how the system should react to changes in system behaviour and has no mechanism for long-term planning of system changes beyond an administrator defined epoch.

6.7 Summary

This review of existing related work reveals that they make only static decisions, are not distributed DBMSs, or do not make integrated decisions about how and where to store data adaptively, resulting in sub-par performance. Therefore, the ideas and techniques presented in this thesis are novel and useful.

Chapter 7

Conclusion

Distributed database systems store vast amounts of data that enterprises utilize to support their organizational needs. Efficiently storing and accessing this data is essential as business-critical applications use database systems as key components. For example, an e-commerce company must process and record sales, analyze these sales for fraud, and persist these records for legal compliance. These critical business requirements are latency-sensitive; for example, increasing the latency of processing a sale increases the likelihood that a customer will not complete their purchase [19, 115]. Therefore, executing transactions with low latency and high throughput in a distributed database system is important.

Distributed database systems use replication and partitioning to store and distribute data among nodes. Data storage and placement decisions significantly affect system performance, affecting data access and retrieval methods. However, the performance advantage for one workload can be detrimental for another workload. Consequently, static data storage and placement decisions suffer in the presence of changing workloads or if the workload cannot be known ahead of time. Therefore, there are significant advantages to making these decisions adaptively. However, managing how and where data is stored adaptively presents significant challenges; the system must execute transactions efficiently while it adapts, the system must model and understand the workload and make prudent storage and placement adaptation decisions. The techniques described in this thesis that enable adaptivity in distributed database systems meet these challenges while improving system performance.

7.1 Contributions

This thesis has explored using adaptation to improve the performance of distributed database systems. I have developed and evaluated three systems in this work: DynaMast, MorphoSys and Proteus, which adapt how and where data is stored. These systems target different workloads, use novel adaptation techniques, and demonstrate the effectiveness of adaptation in distributed databases. These systems share a two-tier architecture that enables adaptation via an adaptation advisor that observes the workload and changes data placement and storage at data sites.

The DynaMast system addressed the performance problems that arise in distributed DBMSs from expensive distributed coordination protocols without routing all transactions to a single master site. DynaMast intelligently decides where to place master copies of data items in a fully replicated architecture, considering factors such as load distribution and access locality. DynaMast adapts these master placement decisions efficiently using the remastering protocol, which leverages replicas to change mastership without stopping and copying the data items. The evaluation of DynaMast shows that it improves throughput by up to 15$\times$ compared to using two-phase commit in a partitioned DBMS. Additionally, the experimental results demonstrate that DynaMast makes effective remastering decisions and that the costs associated with remastering are amortized across transactions.

This thesis also introduces adaptive database physical design for distributed DBMSs in the MorphoSys system. MorphoSys automatically selects and modifies three core aspects of distributed database physical design: how data partitions are formed, where they are replicated, and where the master copy of a partition is located. MorphoSys integrates these three core decisions and makes cost-based design decisions using a learned cost model that predicts the latency of transactions under different physical design decisions. To execute transactions and design changes efficiently in the presence of these adaptations, MorphoSys introduces a partition-based concurrency control and update propagation scheme. The experimental evaluation of MorphoSys shows that automatic adaptation of distributed database physical design precludes the need for static designs requiring prior workload information without sacrificing system performance.

The final research contribution of this thesis targets distributed DBMSs that process hybrid or mixed workloads consisting of both OLTP and OLAP transactions. Traditionally, these database systems store data either entirely in row or columnar format, optimized for either OLTP or OLAP workloads, or keep a complete copy of data in row and columnar format. Consequently, these static decisions either sacrifice performance on one aspect of the hybrid workload or have a high storage overhead. This thesis presented Proteus, which automatically adapts the distributed DBMS storage layout based on the workload, to address these concerns. To do so, Proteus learns the workload by predicting data access costs and when data will be accessed, and selectively and adaptively replicates and partitions data in different storage layouts. Experimental results demonstrate that Proteus reduces workload completion time by up to 70% when compared to static layouts.

The evaluation of these systems demonstrates the effectiveness of automatic adaptation of how and where data is stored in distributed database systems in reducing transactional latency and increasing system throughput. Thus, distributed database systems that must make data storage and placement decisions, can significantly benefit from adapting these decisions based on the workload to improve their performance.

7.2 Future Work

I now discuss some possible directions for future research based on the work presented in this thesis.

7.2.1 Adapting to the Environment

This thesis focused on adapting distributed DBMSs based on the workload submitted by the clients in terms of queries and transactions. However, the environment that the DBMS executes in, including the availability, capability and price of compute, storage, and network, e.g., in the cloud, can change over time [17, 203, 190, 93]. By using similar forecasting techniques as in Proteus, a DBMS could learn the patterns in the availability and pricing of these resources. Combining these predictions with predictions of the future workload, the DBMS could decide when to add or remove resources to execute transactions while efficiently meeting latency objectives.

7.2.2 Geo Distributed DBMSs

In this thesis, I examined adaptivity in the context of distributed DBMS within a local area network, such as a data centre. However, geo-distributed DBMSs are increasingly popular due to the global user-base of organizations [192, 50, 179, 23, 136, 209]. However, geo-distributed DBMSs suffer from the network latency incurred by each round trip of communication across regions. For example, the differences in latency between a transaction that crosses regions compared to within a region is significant. Consequently, geo-distributed DBMSs often require client applications to carefully place master and replica copies of data and design applications with these costs in mind. Automatically altering these choices based on the workload can improve the application's performance. However, the different constraints mean that the techniques described in this thesis, e.g., the design of the adaptation advisor, would require alteration.

7.2.3 Disaggregated Database Architectures

This thesis examines distributed DBMSs with a shared-nothing architecture, wherein each site in the system has a compute, memory, and storage unit, and all communication with other nodes is done over a network.

Disaggregated DBMS architectures allow independent scaling of compute and storage resources based on workload demands and have seen increasing interest [116, 30, 212]. As described in this thesis, accessing local and remote data is explicit in the shared-nothing environment. By contrast, disaggregated architectures can present a uniform storage access interface that retrieves and accesses data across the network. However, disaggregated data is still stored on physical storage nodes that based on their physical location, can induce locality on the compute nodes [210]. Hence, automatically adapting how and where data is stored in a disaggregated DBMS presents an opportunity to improve the system's performance. Specifically, altering data placement and storage layouts when the system adds or removes compute or storage resources improves storage locality and reduces transaction latency. Similarly, if a system could adapt its physical data storage so that data items co-accessed in transactions could be accessed in a single remote direct memory access (RDMA), then the number of remote accesses could be minimized.

7.3 Concluding Remarks

Organizations rely on large amounts of data to make data-driven decisions. The continued growth of e-commerce transactions, machine learning pipelines, and data warehousing solutions will only continue to increase these demands. Organizations rely on distributed database management systems to efficiently store and manage this data. Managing this distributed data requires making data storage and placement decisions, affecting system performance. Static data storage and placement decisions cease to be effective in the presence of workload changes. In this thesis, I have focused on improving the performance of distributed database systems by automatically adapting how and where data is stored. The evaluation of the systems presented in this thesis shows that the techniques effectively improve database system performance. I believe that the research contributions in this thesis will pave the way for developing and adopting adaptive and self-driving distributed database systems.

References

[1] The Transaction Processing Council. TPC-C Benchmark (Revision 5.11)., 2010.

[2] Nuodb architecture. http://go.nuodb.com/rs/nuodb/images/Technical-Whitepaper.pdf, 2017. Accessed: 2018-12-07.

[3] Apache parquet, 2018.

[4] The Transaction Processing Council. TPC-H Benchmark (Revision 2.18)., 2018.

[5] Huawei relational database service. https://support.huaweicloud.com/en-us/productdesc-rds/rds-productdesc.pdf, 2019. Accessed: 2019-06-01.

[6] Mysql: Primary-secondary replication. https://dev.mysql.com/doc/refman/8.0/en/group-replication-primary-secondary-replication.html, 2019. Accessed: 2019-02-01.

[7] Api reference index — twitter api, 2021.

[8] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In Surajit Chaudhuri, Vagelis Hristidis, and Neoklis Polyzotis, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, pages 671–682. ACM, 2006.

[9] Daniel J. Abadi, Samuel Madden, and Nabil Hachem. Column-stores vs. row-stores: how different are they really? In Jason Tsong-Li Wang, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 967–980. ACM, 2008.

[10] Daniel J. Abadi, Daniel S. Myers, David J. DeWitt, and Samuel Madden. Materialization strategies in a column-oriented DBMS. In Rada Chirkova, Asuman Dogac, M. Tamer Özsu, and Timos K. Sellis, editors, *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, pages 466–475. IEEE Computer Society, 2007.

[11] Michael Abebe, Khuzaima Daudjee, Brad Glasbergen, and Yuanfeng Tian. EC-Store: Bridging the gap between storage and latency in distributed erasure coded systems. In *38th IEEE International Conference on Distributed Computing Systems, ICDCS 2018, Vienna, Austria, July 2-6, 2018*, pages 255–266. IEEE Computer Society, 2018.

[12] Michael Abebe, Brad Glasbergen, and Khuzaima Daudjee. DynaMast: Adaptive dynamic mastering for replicated systems. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*, pages 1381–1392. IEEE, 2020.

[13] Michael Abebe, Brad Glasbergen, and Khuzaima Daudjee. MorphoSys: Automatic physical design metamorphosis for distributed database systems. *Proc. VLDB Endow.*, 13(13):3573–3587, 2020.

[14] Michael Abebe, Horatiu Lazu, and Khuzaima Daudjee. Proteus: Autonomous adaptive storage for mixed workloads. In Zachary Ives, Angela Bonifati, and Amr El Abbadi, editors, *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, pages 700–714. ACM, 2022.

[15] Michael Abebe, Horatiu Lazu, and Khuzaima Daudjee. Tiresias: Enabling predictive autonomous storage and indexing. *Proc. VLDB Endow.*, 15(11):3126–3136, 2022.

[16] Atul Adya, Barbara Liskov, and Patrick E. O'Neil. Generalized isolation level definitions. In David B. Lomet and Gerhard Weikum, editors, *Proceedings of the 16th International Conference on Data Engineering, San Diego, California, USA, February 28 - March 3, 2000*, pages 67–78. IEEE Computer Society, 2000.

[17] Divyakant Agrawal, Sudipto Das, and Amr El Abbadi. Big data and cloud computing: current state and future opportunities. In Anastasia Ailamaki, Sihem Amer-Yahia, Jignesh M. Patel, Tore Risch, Pierre Senellart, and Julia Stoyanovich, editors, *EDBT 2011, 14th International Conference on Extending Database Technology, Uppsala, Sweden, March 21-24, 2011, Proceedings*, pages 530–533. ACM, 2011.

[18] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollár, Arunprasad P. Marathe, Vivek R. Narasayya, and Manoj Syamala. Database tuning advisor for microsoft SQL server

2005. In Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer, editors, *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004*, pages 1110–1121. Morgan Kaufmann, 2004.

[19] Akamai. New study reveals the impact of travel site performance on consumers. https://www.akamai.com/us/en/about/news/press/2010-press/new-study-reveals-the-impact-of-travel-site-performance-on-consumers.jsp, 2010.

[20] Dana Van Aken, Andrew Pavlo, Geoffrey J. Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1009–1024. ACM, 2017.

[21] Ioannis Alagiannis, Stratos Idreos, and Anastasia Ailamaki. H2O: a hands-free adaptive store. In Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu, editors, *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 1103–1114. ACM, 2014.

[22] Ahmed Alquraan, Alex Kogan, Virendra J. Marathe, and Samer Al-Kiswany. Scalable, nearzero loss disaster recovery for distributed data stores. *Proc. VLDB Endow.*, 13(9):1429–1442, 2020.

[23] Krishna Aravind and Mimi Gentz. Multi-master globally replicated database architectures with azure cosmos db. https://docs.microsoft.com/en-us/azure/cosmos-db/multi-region-writers, 2017.

[24] Masoud Saeida Ardekani and Douglas B. Terry. A self-configurable geo-replicated cloud storage system. In Jason Flinn and Hank Levy, editors, *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, pages 367–381. USENIX Association, 2014.

[25] Vaibhav Arora, Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. Janus: A hybrid scalable multi-representation cloud datastore. *IEEE Trans. Knowl. Data Eng.*, 30(4):689–702, 2018.

[26] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. Bridging the archipelago between row-stores and column-stores for hybrid workloads. In Fatma Özcan, Georgia

Koutrika, and Sam Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 583–598. ACM, 2016.

[27] Manos Athanassoulis, Kenneth S. Bøgh, and Stratos Idreos. Optimal column layout for hybrid workloads. *Proc. VLDB Endow.*, 12(13):2393–2407, 2019.

[28] Peter Bailis, Alan D. Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Coordination avoidance in database systems. *Proc. VLDB Endow.*, 8(3):185–196, 2014.

[29] Peter Bailis, Edward Gan, Samuel Madden, Deepak Narayanan, Kexin Rong, and Sahaana Suri. Macrobase: Prioritizing attention in fast data. In Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 541–556. ACM, 2017.

[30] Tiemo Bang, Norman May, Ilia Petrov, and Carsten Binnig. Anydb: An architecture-less DBMS for any workload. In *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings.* www.cidrdb.org, 2021.

[31] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. A critique of ANSI SQL isolation levels. In Michael J. Carey and Donovan A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA, May 22-25, 1995*, pages 1–10. ACM Press, 1995.

[32] Carsten Binnig, Stefan Hildenbrand, Franz Färber, Donald Kossmann, Juchang Lee, and Norman May. Distributed snapshot isolation: global transactions pay globally, local transactions pay locally. *VLDB J.*, 23(6):987–1011, 2014.

[33] Mike W. Blasgen, Morton M. Astrahan, Donald D. Chamberlin, Jim Gray, W. Frank King III, Bruce G. Lindsay, Raymond A. Lorie, James W. Mehl, Thomas G. Price, Gianfranco R. Putzolu, Mario Schkolnick, Patricia G. Selinger, Donald R. Slutz, H. Raymond Strong, Irving L. Traiger, Bradford W. Wade, and Robert A. Yost. System R: an architectural overview. *IBM Syst. J.*, 20(1):41–62, 1981.

[34] Peter A. Boncz, Marcin Zukowski, and Niels Nes. Monetdb/x100: Hyper-pipelining query execution. In *Second Biennial Conference on Innovative Data Systems Re-*

search, CIDR 2005, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings, pages 225–237. www.cidrdb.org, 2005.

[35] Mihaela A. Bornea, Orion Hodson, Sameh Elnikety, and Alan D. Fekete. One-copy serializability with snapshot isolation under the hood. In Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan, editors, *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 625–636. IEEE Computer Society, 2011.

[36] Christof Bornhövd, Mehmet Altinel, Sailesh Krishnamurthy, C. Mohan, Hamid Pirahesh, and Berthold Reinwald. Dbcache: Middle-tier database caching for highly scalable e-business architectures. In Alon Y. Halevy, Zachary G. Ives, and AnHai Doan, editors, *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, page 662. ACM, 2003.

[37] Ivan T. Bowman and Kenneth Salem. Optimization of query streams using semantic prefetching. *ACM Trans. Database Syst.*, 30(4):1056–1101, 2005.

[38] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry C. Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkateshwaran Venkataramani. TAO: facebook's distributed data store for the social graph. In Andrew Birrell and Emin Gün Sirer, editors, *2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013*, pages 49–60. USENIX Association, 2013.

[39] Prima Chairunnanda, Khuzaima Daudjee, and M. Tamer Özsu. Confluxdb: Multi-master replication for partitioned snapshot isolation databases. *Proc. VLDB Endow.*, 7(11):947–958, 2014.

[40] Wilson Wai Shun Chan et al. Techniques for multiple window resource remastering among nodes of a cluster, 2008. US Patent 7,379,952.

[41] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A distributed storage system for structured data. In Brian N. Bershad and Jeffrey C. Mogul, editors, *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, pages 205–218. USENIX Association, 2006.

[42] Surajit Chaudhuri and Vivek R. Narasayya. Autoadmin 'what-if' index analysis utility. In Laura M. Haas and Ashutosh Tiwary, editors, *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 367–378. ACM Press, 1998.

[43] Gong Chen, Wenbo He, Jie Liu, Suman Nath, Leonidas Rigas, Lin Xiao, and Feng Zhao. Energy-aware server provisioning and load dispatching for connection-intensive internet services. In Jon Crowcroft and Michael Dahlin, editors, *5th USENIX Symposium on Networked Systems Design & Implementation, NSDI 2008, April 16-18, 2008, San Francisco, CA, USA, Proceedings*, pages 337–350. USENIX Association, 2008.

[44] Haibo Chen, Rong Chen, Xingda Wei, Jiaxin Shi, Yanzhe Chen, Zhaoguo Wang, Binyu Zang, and Haibing Guan. Fast in-memory transaction processing using RDMA and HTM. *ACM Trans. Comput. Syst.*, 35(1):3:1–3:37, 2017.

[45] Edgar F Codd. Is your dbms really relational. *ComputerWorld*, 14(10), 1985.

[46] Richard L. Cole, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, Harumi A. Kuno, Raghunath Othayoth Nambiar, Thomas Neumann, Meikel Poess, Kai-Uwe Sattler, Michael Seibold, Eric Simon, and Florian Waas. The mixed workload ch-benchmark. In Goetz Graefe and Kenneth Salem, editors, *Proceedings of the Fourth International Workshop on Testing Database Systems, DBTest 2011, Athens, Greece, June 13, 2011*, page 8. ACM, 2011.

[47] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, 2008.

[48] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In Joseph M. Hellerstein, Surajit Chaudhuri, and Mendel Rosenblum, editors, *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, pages 143–154. ACM, 2010.

[49] George P. Copeland and Setrag Khoshafian. A decomposition storage model. In Shamkant B. Navathe, editor, *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data, Austin, Texas, USA, May 28-31, 1985*, pages 268–279. ACM Press, 1985.

[50] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In Chandu Thekkath and Amin Vahdat, editors, *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 251–264. USENIX Association, 2012.

[51] James A. Cowling and Barbara Liskov. Granola: Low-overhead distributed transaction coordination. In Gernot Heiser and Wilson C. Hsieh, editors, *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, pages 223–235. USENIX Association, 2012.

[52] Natacha Crooks, Youer Pu, Lorenzo Alvisi, and Allen Clement. Seeing is believing: A client-centric specification of database isolation. In Elad Michael Schiller and Alexander A. Schwarzmann, editors, *Proceedings of the ACM Symposium on Principles of Distributed Computing, PODC 2017, Washington, DC, USA, July 25-27, 2017*, pages 73–82. ACM, 2017.

[53] Carlo Curino, Yang Zhang, Evan P. C. Jones, and Samuel Madden. Schism: a workload-driven approach to database replication and partitioning. *Proc. VLDB Endow.*, 3(1):48–57, 2010.

[54] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. G-store: a scalable data store for transactional multi key access in the cloud. In Joseph M. Hellerstein, Surajit Chaudhuri, and Mendel Rosenblum, editors, *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, pages 163–174. ACM, 2010.

[55] Sudipto Das, Miroslav Grbic, Igor Ilic, Isidora Jovandic, Andrija Jovanovic, Vivek R. Narasayya, Miodrag Radulovic, Maja Stikic, Gaoxiang Xu, and Surajit Chaudhuri. Automatically indexing millions of databases in microsoft azure SQL database. In Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 666–679. ACM, 2019.

[56] Khuzaima Daudjee and Kenneth Salem. Lazy database replication with snapshot isolation. In Umeshwar Dayal, Kyu-Young Whang, David B. Lomet, Gustavo Alonso, Guy M. Lohman, Martin L. Kersten, Sang Kyun Cha, and Young-Kuk Kim, editors, *Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006*, pages 715–726. ACM, 2006.

[57] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: SQL server's memory-optimized OLTP engine. In Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 1243–1254. ACM, 2013.

[58] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proc. VLDB Endow.*, 7(4):277–288, 2013.

[59] Jialin Ding, Umar Farooq Minhas, Badrish Chandramouli, Chi Wang, Yinan Li, Ying Li, Donald Kossmann, Johannes Gehrke, and Tim Kraska. Instance-optimized data layouts for cloud analytics workloads. In Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 418–431. ACM, 2021.

[60] Aleksandar Dragojevic, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: distributed transactions with consistency, availability, and performance. In Ethan L. Miller and Steven Hand, editors, *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 54–70. ACM, 2015.

[61] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. Tuning database configuration parameters with ituned. *Proc. VLDB Endow.*, 2(1):1246–1257, 2009.

[62] Aaron J. Elmore, Vaibhav Arora, Rebecca Taft, Andrew Pavlo, Divyakant Agrawal, and Amr El Abbadi. Squall: Fine-grained live reconfiguration for partitioned main memory databases. In Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 299–313. ACM, 2015.

[63] Yaakov Engel, Shie Mannor, and Ron Meir. The kernel recursive least-squares algorithm. *IEEE Trans. Signal Process.*, 52(8):2275–2285, 2004.

[64] Elias Escobedo. The impact of hyped ipo's on the market. Technical report, University of Northern Iowa, 2015.

[65] Brad Glasbergen, Michael Abebe, Khuzaima Daudjee, and Amit Levi. Sentinel: Universal analysis and insight for data systems. *Proc. VLDB Endow.*, 13(11):2720–2733, 2020.

[66] Brad Glasbergen, Kyle Langendoen, Michael Abebe, and Khuzaima Daudjee. Chronocache: Predictive and adaptive mid-tier query result caching. In David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 2391–2406. ACM, 2020.

[67] Brad Glasbergen, Fangyu Wu, and Khuzaima Daudjee. Dendrite: Bolt-on adaptivity for data systems. In Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 2726–2730. ACM, 2021.

[68] Anil K. Goel, Jeffrey Pound, Nathan Auch, Peter Bumbulis, Scott MacLean, Franz Färber, Francis Gropengießer, Christian Mathis, Thomas Bodner, and Wolfgang Lehner. Towards scalable real-time analytics: An architecture for scale-out of olxp workloads. *Proc. VLDB Endow.*, 8(12):1716–1727, 2015.

[69] Cary G. Gray and David R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In Gregory R. Andrews, editor, *Proceedings of the Twelfth ACM Symposium on Operating System Principles, SOSP 1989, The Wigwam, Litchfield Park, Arizona, USA, December 3-6, 1989*, pages 202–210. ACM, 1989.

[70] Jim Gray, Pat Helland, Patrick E. O'Neil, and Dennis E. Shasha. The dangers of replication and a solution. In H. V. Jagadish and Inderpal Singh Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, pages 173–182. ACM Press, 1996.

[71] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudré-Mauroux, and Samuel Madden. HYRISE - A main memory hybrid storage engine. *Proc. VLDB Endow.*, 4(2):105–116, 2010.

[72] Himanshu Gupta. Selection of views to materialize in a data warehouse. In Foto N. Afrati and Phokion G. Kolaitis, editors, *Database Theory - ICDT '97, 6th International Conference, Delphi, Greece, January 8-10, 1997, Proceedings*, volume 1186 of *Lecture Notes in Computer Science*, pages 98–112. Springer, 1997.

[73] Ramesh Gupta, Jayant R. Haritsa, and Krithi Ramamritham. Revisiting commit processing in distributed database systems. In Joan Peckham, editor, *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*, pages 486–497. ACM Press, 1997.

[74] Felix Halim, Stratos Idreos, Panagiotis Karras, and Roland H. C. Yap. Stochastic database cracking: Towards robust adaptive indexing in main-memory column-stores. *Proc. VLDB Endow.*, 5(6):502–513, 2012.

[75] Richard A. Hankins and Jignesh M. Patel. Data morphing: An adaptive, cache-conscious storage technique. In Johann Christoph Freytag, Peter C. Lockemann, Serge Abiteboul, Michael J. Carey, Patricia G. Selinger, and Andreas Heuer, editors, *Proceedings of 29th International Conference on Very Large Data Bases, VLDB 2003, Berlin, Germany, September 9-12, 2003*, pages 417–428. Morgan Kaufmann, 2003.

[76] Rachael Harding, Dana Van Aken, Andrew Pavlo, and Michael Stonebraker. An evaluation of distributed concurrency control. *Proc. VLDB Endow.*, 10(5):553–564, 2017.

[77] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. OLTP through the looking glass, and what we found there. In Jason Tsong-Li Wang, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 981–992. ACM, 2008.

[78] Trevor Hastie, Jerome H. Friedman, and Robert Tibshirani. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction.* Springer Series in Statistics. Springer, 2001.

[79] Herodotos Herodotou and Elena Kakoulli. Automating distributed tiered storage management in cluster computing. *Proc. VLDB Endow.*, 13(1):43–56, 2019.

[80] Benjamin Hilprecht, Carsten Binnig, and Uwe Röhm. Learning a partitioning advisor for cloud databases. In David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew

Tan, Abdussalam Alawini, and Hung Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 143–157. ACM, 2020.

[81] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, 1997.

[82] Yu-Ju Hong and Mithuna Thottethodi. Understanding and mitigating the impact of load imbalance in the memory caching tier. In Guy M. Lohman, editor, *ACM Symposium on Cloud Computing, SOCC '13, Santa Clara, CA, USA, October 1-3, 2013*, pages 13:1–13:17. ACM, 2013.

[83] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. Erasure coding in windows azure storage. In Gernot Heiser and Wilson C. Hsieh, editors, *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*, pages 15–26. USENIX Association, 2012.

[84] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. Tidb: A raft-based HTAP database. *Proc. VLDB Endow.*, 13(12):3072–3084, 2020.

[85] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. X-engine: An optimized storage engine for large-scale e-commerce transaction processing. In Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 651–665. ACM, 2019.

[86] Qi Huang, Ken Birman, Robbert van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C. Li. An analysis of facebook photo caching. In Michael Kaminsky and Mike Dahlin, editors, *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, pages 167–181. ACM, 2013.

[87] Yihe Huang, William Qian, Eddie Kohler, Barbara Liskov, and Liuba Shrira. Opportunities for optimism in contended main-memory multicore transactions. *Proc. VLDB Endow.*, 13(5):629–642, 2020.

[88] Paul Hudak. A semantic model of reference counting and its abstraction (detailed summary). In William L. Scherlis, John H. Williams, and Richard P. Gabriel, editors, *Proceedings of the 1986 ACM Conference on LISP and Functional Programming, LFP 1986, Cambridge, Massachusetts, USA, August 4-6, 1986*, pages 351–363. ACM, 1986.

[89] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Database cracking. In *Third Biennial Conference on Innovative Data Systems Research, CIDR 2007, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings*, pages 68–78. www.cidrdb.org, 2007.

[90] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Self-organizing tuple reconstruction in column-stores. In Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pages 297–308. ACM, 2009.

[91] Stratos Idreos, Stefan Manegold, Harumi A. Kuno, and Goetz Graefe. Merging what's cracked, cracking what's merged: Adaptive indexing in main-memory column-stores. *Proc. VLDB Endow.*, 4(9):585–597, 2011.

[92] Alekh Jindal. The mimicking octopus: Towards a one-size-fits-all database architecture. In *VLDB PhD Workshop*, pages 78–83, 2010.

[93] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, João Carreira, Karl Krauth, Neeraja Jayant Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud programming simplified: A berkeley view on serverless computing. *CoRR*, abs/1902.03383, 2019.

[94] Evan P. C. Jones, Daniel J. Abadi, and Samuel Madden. Low overhead concurrency control for partitioned main memory databases. In Ahmed K. Elmagarmid and Divyakant Agrawal, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*, pages 603–614. ACM, 2010.

[95] James B. Rothnie Jr., Philip A. Bernstein, Stephen Fox, Nathan Goodman, Michael Hammer, Terry A. Landers, Christopher L. Reeve, David W. Shipman, and Eugene Wong. Introduction to a system for distributed databases (SDD-1). *ACM Trans. Database Syst.*, 5(1):1–17, 1980.

[96] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alex Rasin, Stanley B. Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.*, 1(2):1496–1499, 2008.

[97] Donghe Kang, Ruochen Jiang, and Spyros Blanas. Jigsaw: A data storage and query processing engine for irregular table partitioning. In Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 898–911. ACM, 2021.

[98] George Karypis. Metis: Unstructured graph partitioning and sparse matrix ordering system. *Technical report*, 1997.

[99] Alfons Kemper and Thomas Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan, editors, *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 195–206. IEEE Computer Society, 2011.

[100] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter A. Boncz. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *Proc. VLDB Endow.*, 11(13):2209–2222, 2018.

[101] Davis E. King. Dlib-ml: A machine learning toolkit. *J. Mach. Learn. Res.*, 10:1755–1758, 2009.

[102] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, volume 11, pages 1–7, 2011.

[103] Konstantinos Krikellas, Sameh Elnikety, Zografoula Vagena, and Orion Hodson. Strongly consistent replication for a bargain. In Feifei Li, Mirella M. Moro, Shahram Ghandeharizadeh, Jayant R. Haritsa, Gerhard Weikum, Michael J. Carey, Fabio Casati, Edward Y. Chang, Ioana Manolescu, Sharad Mehrotra, Umeshwar Dayal, and Vassilis J. Tsotras, editors, *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*, pages 52–63. IEEE Computer Society, 2010.

[104] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph M. Hellerstein, and Ion Stoica. Learning to optimize join queries with deep reinforcement learning. *CoRR*, abs/1808.03196, 2018.

[105] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. High-performance concurrency control mechanisms for main-memory databases. *Proc. VLDB Endow.*, 5(4):298–309, 2011.

[106] Per-Åke Larson, Cipri Clinciu, Campbell Fraser, Eric N. Hanson, Mostafa Mokhtar, Michal Nowakiewicz, Vassilis Papadimos, Susan L. Price, Srikumar Rangarajan, Remus Rusanu, and Mayukh Saubhasik. Enhancements to SQL server column stores. In Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 1159–1168. ACM, 2013.

[107] Per-Åke Larson, Jonathan Goldstein, Hongfei Guo, and Jingren Zhou. Mtcache: Mid-tier database caching for SQL server. *IEEE Data Eng. Bull.*, 27(2):35–40, 2004.

[108] Per-Åke Larson, Eric N. Hanson, and Susan L. Price. Columnar storage in SQL server 2012. *IEEE Data Eng. Bull.*, 35(1):15–20, 2012.

[109] Jonathan Ledlie and Margo I. Seltzer. Distributed, secure load balancing with skew, heterogeneity and churn. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies, 13-17 March 2005, Miami, FL, USA*, pages 1419–1430. IEEE, 2005.

[110] Viktor Leis, Alfons Kemper, and Thomas Neumann. Scaling htm-supported database transactions to many cores. *IEEE Trans. Knowl. Data Eng.*, 28(2):297–310, 2016.

[111] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno M. Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In Chandu Thekkath and Amin Vahdat, editors, *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 265–278. USENIX Association, 2012.

[112] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. Just say NO to paxos overhead: Replacing consensus with network ordering. In Kimberly Keeton and Timothy Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 467–483. USENIX Association, 2016.

[113] Qian Lin, Pengfei Chang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Zhengkui Wang. Towards a non-2pc transaction management in distributed database systems. In Fatma Özcan, Georgia Koutrika, and Sam Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1659–1674. ACM, 2016.

[114] Qian Lin, Pengfei Chang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Zhengkui Wang. Towards a non-2pc transaction management in distributed database systems. In Fatma Özcan, Georgia Koutrika, and Sam Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1659–1674. ACM, 2016.

[115] Greg Linden. Make data useful. http://www.gduchamp.com/media/StanfordDataMining.2006-11-28.pdf.

[116] Simon Loesing, Markus Pilman, Thomas Etter, and Donald Kossmann. On the design and scalability of distributed shared-data databases. In Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 663–676. ACM, 2015.

[117] Guy M. Lohman, C. Mohan, Laura M. Haas, Dean Daniels, Bruce G. Lindsay, Patricia G. Selinger, and Paul F. Wilms. Query processing in r*. In Won Kim, David S. Reiner, and Don S. Batory, editors, *Query Processing in Database Systems*, pages 31–47. Springer, 1985.

[118] Yi Lu, Anil Shanbhag, Alekh Jindal, and Samuel Madden. Adaptdb: Adaptive partitioning for distributed joins. *Proc. VLDB Endow.*, 10(5):589–600, 2017.

[119] Yi Lu, Xiangyao Yu, and Samuel Madden. STAR: scaling transactions through asymmetric replication. *Proc. VLDB Endow.*, 12(11):1316–1329, 2019.

[120] Zhenghua Lyu, Huan Hubert Zhang, Gang Xiong, Gang Guo, Haozhou Wang, Jinbao Chen, Asim Praveen, Yu Yang, Xiaoming Gao, Alexandra Wang, Wen Lin, Ashwin Agrawal, Junfeng Yang, Hao Wu, Xiaoliang Li, Feng Guo, Jiang Wu, Jesse Zhang, and Venkatesh Raghavan. Greenplum: A hybrid database for transactional and analytical workloads. In Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 2530–2542. ACM, 2021.

[121] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J. Gordon. Query-based workload forecasting for self-driving database management systems. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 631–645. ACM, 2018.

[122] Sam Madden. From databases to big data. *IEEE Internet Comput.*, 16(3):4–6, 2012.

[123] Ashraf Mahgoub, Paul Wood, Sachandhan Ganesh, Subrata Mitra, Wolfgang Gerlach, Travis Harrison, Folker Meyer, Ananth Grama, Saurabh Bagchi, and Somali Chaterji. Rafiki: a middleware for parameter tuning of nosql datastores for dynamic metagenomics workloads. In K. R. Jayaram, Anshul Gandhi, Bettina Kemme, and Peter R. Pietzuch, editors, *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference, Las Vegas, NV, USA, December 11 - 15, 2017*, pages 28–40. ACM, 2017.

[124] Hatem A. Mahmoud, Vaibhav Arora, Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. Maat: Effective and scalable coordination of distributed transactions in the cloud. *Proc. VLDB Endow.*, 7(5):329–340, 2014.

[125] Darko Makreshanski, Jana Giceva, Claude Barthels, and Gustavo Alonso. Batchdb: Efficient isolated execution of hybrid OLTP+OLAP workloads for interactive applications. In Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 37–50. ACM, 2017.

[126] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. Bao: Making learned query optimization practical. In Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 1275–1288. ACM, 2021.

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

[128] Ryan Marcus, Olga Papaemmanouil, Sofiya Semenova, and Solomon Garber. Nashdb: An end-to-end economic method for elastic database fragmentation, replication, and

provisioning. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1253–1267. ACM, 2018.

[129] Ryan C. Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. Neo: A learned query optimizer. *Proc. VLDB Endow.*, 12(11):1705–1718, 2019.

[130] Ryan C. Marcus and Olga Papaemmanouil. Plan-structured deep neural network models for query performance prediction. *Proc. VLDB Endow.*, 12(11):1733–1746, 2019.

[131] B. Scott Michel, Khoi Nguyen, Adam Rosenstein, Lixia Zhang, Sally Floyd, and Van Jacobson. Adaptive web caching: towards a new global caching architecture. *Comput. Networks*, 30(22-23):2169–2177, 1998.

[132] James Mickens. The night watch. *login: the USENIX magazine*, pages 5–7, 2013.

[133] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. Consolidating concurrency control and consensus for commits under conflicts. In Kimberly Keeton and Timothy Roscoe, editors, *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, pages 517–532. USENIX Association, 2016.

[134] Niloy Mukherjee, Shasank Chavan, Maria Colgan, Dinesh Das, Mike Gleeson, Sanket Hase, Allison Holloway, Hui Jin, Jesse Kamp, Kartik Kulkarni, Tirthankar Lahiri, Juan Loaiza, Neil MacNaughton, Vineet Marwah, Atrayee Mullick, Andy Witkowski, Jiaqi Yan, and Mohamed Zaït. Distributed architecture of oracle database in-memory. *Proc. VLDB Endow.*, 8(12):1630–1641, 2015.

[135] Peter Muth, Thomas C. Rakow, Gerhard Weikum, Peter Brössler, and Christof Hasse. Semantic concurrency control in object-oriented database systems. In *Proceedings of the Ninth International Conference on Data Engineering, April 19-23, 1993, Vienna, Austria*, pages 233–242. IEEE Computer Society, 1993.

[136] Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. The challenges of global-scale data management. In Fatma Özcan, Georgia Koutrika, and Sam Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 2223–2227. ACM, 2016.

[137] Alex Nazaruk and Michael Rauchman. Big data in capital markets. In Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 917–918. ACM, 2013.

[138] Thomas Neumann and Michael J. Freitag. Umbra: A disk-based system with in-memory performance. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020.

[139] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In Nick Feamster and Jeffrey C. Mogul, editors, *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*, pages 385–398. USENIX Association, 2013.

[140] Matthaios Olma, Manos Karpathiotakis, Ioannis Alagiannis, Manos Athanassoulis, and Anastasia Ailamaki. Slalom: Coasting through raw data via adaptive partitioning and indexing. *Proc. VLDB Endow.*, 10(10):1106–1117, 2017.

[141] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The LRU-K page replacement algorithm for database disk buffering. In Peter Buneman and Sushil Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 26-28, 1993*, pages 297–306. ACM Press, 1993.

[142] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S. Sathiya Keerthi. Learning state representations for query optimization with deep reinforcement learning. In Sebastian Schelter, Stephan Seufert, and Arun Kumar, editors, *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning, DEEM@SIGMOD 2018, Houston, TX, USA, June 15, 2018*, pages 4:1–4:4. ACM, 2018.

[143] Adrian Otto. Shopify and google cloud team up for an epic bfcm weekend, 2021.

[144] Fatma Özcan, Yuanyuan Tian, and Pinar Tözün. Hybrid transactional/analytical processing: A survey. In Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu, editors, *Proceedings of the 2017 ACM International Conference on*

Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017, pages 1771–1775. ACM, 2017.

[145] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems, 4th Edition*. Springer, 2020.

[146] Vinit Padhye, Gowtham Rajappan, and Anand R. Tripathi. Transaction management using causal snapshot isolation in partially replicated databases. In *33rd IEEE International Symposium on Reliable Distributed Systems, SRDS 2014, Nara, Japan, October 6-9, 2014*, pages 105–114. IEEE Computer Society, 2014.

[147] Vinit Padhye and Anand Tripathi. Causally coordinated snapshot isolation for geographically replicated data. In *IEEE 31st Symposium on Reliable Distributed Systems, SRDS 2012, Irvine, CA, USA, October 8-11, 2012*, pages 261–266. IEEE Computer Society, 2012.

[148] Mark Palmer and Stanley B. Zdonik. Fido: A cache that learns to fetch. In Guy M. Lohman, Amílcar Sernadas, and Rafael Camps, editors, *17th International Conference on Very Large Data Bases, September 3-6, 1991, Barcelona, Catalonia, Spain, Proceedings*, pages 255–264. Morgan Kaufmann, 1991.

[149] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 8024–8035, 2019.

[150] Andrew Pavlo, Carlo Curino, and Stanley B. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In K. Selçuk Candan, Yi Chen, Richard T. Snodgrass, Luis Gravano, and Ariel Fuxman, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 61–72. ACM, 2012.

[151] Andy Pavlo. Make your database system dream of electric sheep: Towards self-driving operation. *Proc. VLDB Endow.*, 14(12):3211–3221, 2021.

[152] M Pezzini, D Feinberg, N Rayner, and R Edjlali. Real-time insights and decision making using hybrid streaming, in-memory computing analytics and transaction processing. *Gartner*, 2016.

[153] Massimo Pezzini, Donald Feinberg, Nigel Rayner, and Roxane Edjlali. Hybrid transaction/analytical processing will foster opportunities for dramatic business innovation. *Gartner*, pages 4–20, 2014.

[154] Yiming Qian, Xinjian Shao, and Jingchi Liao. Pre-ipo hype by affiliated analysts: Motives and consequences. *Social Science Research Network*, 2019.

[155] Abdul Quamar, K. Ashwin Kumar, and Amol Deshpande. SWORD: scalable workload-aware data placement for transactional workloads. In Giovanna Guerrini and Norman W. Paton, editors, *Joint 2013 EDBT/ICDT Conferences, EDBT '13 Proceedings, Genoa, Italy, March 18-22, 2013*, pages 430–441. ACM, 2013.

[156] Tilmann Rabl and Hans-Arno Jacobsen. Query centric partitioning and allocation for partially replicated database systems. In Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 315–330. ACM, 2017.

[157] Raghu Ramakrishnan and Johannes Gehrke. *Database management systems (3.ed.)*. McGraw-Hill, 2003.

[158] Ravishankar Ramamurthy, David J. DeWitt, and Qi Su. A case for fractured mirrors. *VLDB J.*, 12(2):89–101, 2003.

[159] Jacob Ratkiewicz, Santo Fortunato, Alessandro Flammini, Filippo Menczer, and Alessandro Vespignani. Characterizing and modeling the dynamics of online popularity. *Physical review letters*, 105(15):158701, 2010.

[160] Aunn Raza, Periklis Chrysogelos, Angelos-Christos G. Anadiotis, and Anastasia Ailamaki. Adaptive HTAP through elastic resource scheduling. In David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 2043–2054. ACM, 2020.

[161] P. Krishna Reddy and Masaru Kitsuregawa. Speculative locking protocols to improve performance for distributed database system. *IEEE Trans. Knowl. Data Eng.*, 16(2):154–169, 2004.

[162] Kun Ren, Dennis Li, and Daniel J. Abadi. SLOG: serializable, low-latency, geo-replicated transactions. *Proc. VLDB Endow.*, 12(11):1747–1761, 2019.

[163] Kun Ren, Alexander Thomson, and Daniel J. Abadi. An evaluation of the advantages and disadvantages of deterministic database systems. *Proc. VLDB Endow.*, 7(10):821–832, 2014.

[164] Rodolfo F. Resende, Divyakant Agrawal, and Amr El Abbadi. Semantic locking in object-oriented database systems. In Jeff McKenna, J. Eliot B. Moss, and Richard L. Wexelblat, editors, *Proceedings of the Ninth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA 1994, Portland, Oregon, USA, October 23-27, 1994*, pages 388–402. ACM, 1994.

[165] David Rolfe. Voltdb and ycsb. http://www.voltdb.com/blog/2019/10/16/voltdb-and-ycsb/, 2019. Accessed: 2019-12-16.

[166] Francisco Romero, Gohar Irfan Chaudhry, Iñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J. Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. Faa$t: A transparent auto-scaling cache for serverless applications. In Carlo Curino, Georgia Koutrika, and Ravi Netravali, editors, *SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, November 1 - 4, 2021*, pages 122–137. ACM, 2021.

[167] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[168] Mohammad Sadoghi, Souvik Bhattacherjee, Bishwaranjan Bhattacharjee, and Mustafa Canim. L-store: A real-time OLTP and OLAP system. In Michael H. Böhlen, Reinhard Pichler, Norman May, Erhard Rahm, Shan-Hung Wu, and Katja Hose, editors, *Proceedings of the 21st International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018*, pages 540–551. OpenProceedings.org, 2018.

[169] Felix Martin Schuhknecht, Alekh Jindal, and Jens Dittrich. The uncracked pieces in database cracking. *Proc. VLDB Endow.*, 7(2):97–108, 2013.

[170] Felix Martin Schuhknecht, Alekh Jindal, and Jens Dittrich. An experimental evaluation and analysis of database cracking. *VLDB J.*, 25(1):27–52, 2016.

[171] Marco Serafini, Essam Mansour, Ashraf Aboulnaga, Kenneth Salem, Taha Rafiq, and Umar Farooq Minhas. Accordion: Elastic scalability for database systems supporting distributed transactions. *Proc. VLDB Endow.*, 7(12):1035–1046, 2014.

[172] Marco Serafini, Rebecca Taft, Aaron J. Elmore, Andrew Pavlo, Ashraf Aboulnaga, and Michael Stonebraker. Clay: Fine-grained adaptive partitioning for general database schemas. *Proc. VLDB Endow.*, 10(4):445–456, 2016.

[173] Anil Shanbhag, Alekh Jindal, Yi Lu, and Samuel Madden. Amoeba: A shape changing storage system for big data. *Proc. VLDB Endow.*, 9(13):1569–1572, 2016.

[174] Anil Shanbhag, Alekh Jindal, Samuel Madden, Jorge-Arnulfo Quiané-Ruiz, and Aaron J. Elmore. A robust partitioning scheme for ad-hoc query workloads. In *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24-27, 2017*, pages 229–241. ACM, 2017.

[175] Zechao Shang, Xi Liang, Dixin Tang, Cong Ding, Aaron J. Elmore, Sanjay Krishnan, and Michael J. Franklin. Crocodiledb: Efficient database execution through intelligent deferment. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*. www.cidrdb.org, 2020.

[176] Artyom Sharov, Alexander Shraer, Arif Merchant, and Murray Stokely. Take me to your leader! online optimization of distributed storage configurations. *Proc. VLDB Endow.*, 8(12):1490–1501, 2015.

[177] Avi Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts, Seventh Edition*. McGraw-Hill Book Company, 2020.

[178] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. Thrift: Scalable cross-language services implementation, 2007.

[179] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In Ted Wobber and Peter Druschel, editors, *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, pages 385–400. ACM, 2011.

[180] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. LEO - db2's learning optimizer. In Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass, editors, *VLDB 2001,*

Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy, pages 19–28. Morgan Kaufmann, 2001.

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

[182] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. C-store: A column-oriented DBMS. In Klemens Böhm, Christian S. Jensen, Laura M. Haas, Martin L. Kersten, Per-Åke Larson, and Beng Chin Ooi, editors, *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, pages 553–564. ACM, 2005.

[183] Michael Stonebraker, Sam Madden, and Pradeep Dubey. Intel "big data" science and technology center vision and execution plan. *SIGMOD Rec.*, 42(1):44–49, 2013.

[184] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era (it's time for a complete rewrite). In Christoph Koch, Johannes Gehrke, Minos N. Garofalakis, Divesh Srivastava, Karl Aberer, Anand Deshpande, Daniela Florescu, Chee Yong Chan, Venkatesh Ganti, Carl-Christian Kanne, Wolfgang Klas, and Erich J. Neuhold, editors, *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 1150–1160. ACM, 2007.

[185] Michael Stonebraker and Eric Neuhold. A distributed data base version of ingres., 1976.

[186] Michael Stonebraker and Lawrence A. Rowe. The design of postgres. In Carlo Zaniolo, editor, *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 28-30, 1986*, pages 340–355. ACM Press, 1986.

[187] Michael Stonebraker and Ariel Weisberg. The voltdb main memory DBMS. *IEEE Data Eng. Bull.*, 36(2):21–27, 2013.

[188] Muralidhar Subramanian, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Sivakumar Viswanathan, Linpeng Tang, and Sanjeev Kumar. f4: Facebook's warm BLOB storage system. In Jason Flinn and

Hank Levy, editors, *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, pages 383–398. USENIX Association, 2014.

[189] Yihan Sun, Guy E. Blelloch, Wan Shen Lim, and Andrew Pavlo. On supporting efficient snapshot isolation for hybrid workloads with multi-versioned indexes. *Proc. VLDB Endow.*, 13(2):211–225, 2019.

[190] Rebecca Taft, Nosayba El-Sayed, Marco Serafini, Yu Lu, Ashraf Aboulnaga, Michael Stonebraker, Ricardo Mayerhofer, and Francisco Jose Andrade. P-store: An elastic database system with predictive provisioning. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 205–219. ACM, 2018.

[191] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J. Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. E-store: Fine-grained elastic partitioning for distributed transaction processing. *Proc. VLDB Endow.*, 8(3):245–256, 2014.

[192] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. Cockroachdb: The resilient geo-distributed SQL database. In David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 1493–1509. ACM, 2020.

[193] Sean J Taylor and Benjamin Letham. Forecasting at scale. *The American Statistician*, 72(1):37–45, 2018.

[194] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In K. Selçuk Candan, Yi Chen, Richard T. Snodgrass, Luis Gravano, and Ariel Fuxman, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 1–12. ACM, 2012.

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

[196] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu, editors, *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, pages 1041–1052. ACM, 2017.

[197] Lukas Vogel, Alexander van Renen, Satoshi Imamura, Viktor Leis, Thomas Neumann, and Alfons Kemper. Mosaic: A budget-conscious storage engine for relational database systems. *Proc. VLDB Endow.*, 13(11):2662–2675, 2020.

[198] Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. Query fresh: Log shipping on steroids. *Proc. VLDB Endow.*, 11(4):406–419, 2017.

[199] R. Williams, Dean Daniels, Laura M. Haas, George Lapis, Bruce G. Lindsay, Pui Ng, Ron Obermarck, Patricia G. Selinger, Adrian Walker, Paul F. Wilms, and Robert A. Yost. R*: An overview of the architecture. In Peter Scheuermann, editor, *Proceedings of the Second International Conference on Databases: Improving Database Usability and Responsiveness, June 22-24, 1982, Jerusalem, Israel*, pages 1–27. Academic Press, 1982.

[200] Ronald J. Williams and David Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural Comput.*, 1(2):270–280, 1989.

[201] Ouri Wolfson, Sushil Jajodia, and Yixiu Huang. An adaptive data replication algorithm. *ACM Trans. Database Syst.*, 22(2):255–314, 1997.

[202] Eugene Wong. Retrieving dispersed data from SDD-1: A system for distributed databases. In *Proceedings of the Second Berkeley Workshop on Distributed Data Management and Computer Networks, May 25-27, 1977*, pages 217–235. Technical Information Department, Lawrence Berkeley Laboratory, University of California, Berkeley CA, 1977.

[203] Chenggang Wu, Vikram Sreekanti, and Joseph M. Hellerstein. Autoscaling tiered cloud storage in anna. *Proc. VLDB Endow.*, 12(6):624–638, 2019.

[204] Shan-Hung Wu, Tsai-Yu Feng, Meng-Kai Liao, Shao-Kan Pi, and Yu-Shan Lin. T-part: Partitioning of transactions for forward-pushing in deterministic database systems. In Fatma Özcan, Georgia Koutrika, and Sam Madden, editors, *Proceedings*

of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016, pages 1553–1565. ACM, 2016.

[205] Shuqing Wu and Bettina Kemme. Postgres-r (si): Combining replica control with concurrency control based on snapshot isolation. In *ICDE*, 2005.

[206] Jiacheng Yang, Ian Rae, Jun Xu, Jeff Shute, Zhan Yuan, Kelvin Lau, Qiang Zeng, Xi Zhao, Jun Ma, Ziyang Chen, Yuan Gao, Qilin Dong, Junxiong Zhou, Jeremy Wood, Goetz Graefe, Jeffrey F. Naughton, and John Cieslewicz. F1 lightning: HTAP as a service. *Proc. VLDB Endow.*, 13(12):3313–3325, 2020.

[207] Zongheng Yang, Badrish Chandramouli, Chi Wang, Johannes Gehrke, Yinan Li, Umar Farooq Minhas, Per-Åke Larson, Donald Kossmann, and Rajeev Acharya. Qd-tree: Learning data layouts for big data analytics. In David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 193–208. ACM, 2020.

[208] Xiangyao Yu, Yu Xia, Andrew Pavlo, Daniel Sánchez, Larry Rudolph, and Srinivas Devadas. Sundial: Harmonizing concurrency control and caching in a distributed OLTP database management system. *Proc. VLDB Endow.*, 11(10):1289–1302, 2018.

[209] Victor Zakhary, Faisal Nawab, Divy Agrawal, and Amr El Abbadi. Global-scale placement of transactional data stores. In Michael H. Böhlen, Reinhard Pichler, Norman May, Erhard Rahm, Shan-Hung Wu, and Katja Hose, editors, *Proceedings of the 21st International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018*, pages 385–396. OpenProceedings.org, 2018.

[210] Erfan Zamanian, Carsten Binnig, Tim Kraska, and Tim Harris. The end of a myth: Distributed transaction can scale. *Proc. VLDB Endow.*, 10(6):685–696, 2017.

[211] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building consistent transactions with inconsistent replication. *ACM Trans. Comput. Syst.*, 35(4):12:1–12:37, 2018.

[212] Qizhen Zhang, Yifan Cai, Xinyi Chen, Sebastian Angel, Ang Chen, Vincent Liu, and Boon Thau Loo. Understanding the effect of data center resource disaggregation on production dbmss. *Proc. VLDB Endow.*, 13(9):1568–1581, 2020.

[213] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter A. Boncz. Super-scalar RAM-CPU cache compression. In Ling Liu, Andreas Reuter, Kyu-Young Whang, and Jianjun Zhang, editors, *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*, page 59. IEEE Computer Society, 2006.

APPENDICES

Appendix A

Proofs

The appendix of this thesis formalizes the proofs of the previously described techniques.

A.1 DynaMast Provides Strong Session Snapshot Isolation

I now provide a proof that DynaMast (Chapter 3) supports strong session snapshot isolation (SSSI). To do so, I demonstrate that DynaMast suports snapshot isolation (SI) (Appendix A.1.1) before proving that DynaMast supports strong session SI (Appendix A.1.2).

A.1.1 Dynamic Mastering Snapshot Isolation Level Proof Sketch

I provide a proof that DynaMast provides snapshot isolation (SI).

The proof relies on Properties 1 and 2 (Chapter 2.2) of the redo-log, and the following property that holds within the DynaMast system.

Property 3. *Update transactions can write to only the data items mastered at the site of transaction execution.*

The data sites and adaptation advisor enforce this property (Chapter 3.2.2). The data site guarantees that updates occur only to data items granted ownership to the site by the adaptation advisor. Furthermore, the data site will not `release` ownership of data items while transactions update the items. The adaptation advisor uses mutual exclusion in its remastering protocol to guarantee that it sends only a single `grant` message per data item (Chapter 3.2.2).

I first prove that each data site guarantees that transactions execute locally under SI.

Lemma 1. *Each data site S_i guarantees that transactions execute locally under SI.*

Proof. This proof follows from Chapter 3.4.1. To provide SI, a data site must ensure that when a transaction T begins with a begin timestamp, it will not see any updates by any transaction T_2 that commits after T's begin timestamp. Recall from Algorithm 2, that T is assigned a begin timestamp $tvv_{B(T)}[\,]$. The MVCC protocol (Chapter 3.4.1) guarantees that T reads the most recently committed version of data with version number at most $tvv_{B(T)}[\,]$. Hence any transaction T_2 that committed after T that updated such a data item will have $tvv_{B(T)}[\,] < tvv_{T_2}[\,]$; thus T will not see T_2's update.

However, T will read the state of a transaction T_2 that committed before T's begin timestamp, as $tvv_{T_2}[\,] \leq tvv_{B(T)}[\,]$. T will also read its own uncommitted updates.

When T commits, it will have a commit timestamp more recent than any start or commit timestamp at the site. Recall from Algorithm 2 that if T executes at site S_i and updates data, it will increment $svv_i[i]$ and set its commit timestamp as $tvv_T[\,]$, giving T the largest commit timestamp, and its place in S_i's commit order. T uses write locks to enforce the mutual exclusion of updates to data items. These locks are acquired when the begin timestamp is assigned and released when the commit timestamp is assigned (Algorithm 2). Thus, no other transaction T_2 with an overlapping write set at S_i can acquire these locks until after T commits. Thus there cannot be a transaction T_2 that updates a data item updated by T with overlapping begin and commit timestamps.

Hence the requirements for SI at a data site are satisfied. □

I next prove a lemma that follows from Properties 1 and 2, that defines the order in which refresh transactions are applied at sites.

Lemma 2. *If transaction T_1 commits before T_2 at site S_i, indicated by commit timestamp, then its refresh transaction $R(T_1)$ commits before T_2's refresh transaction $R(T_2)$ at all replica sites S_j. That is, refresh transactions originating from a site are applied in commit order at replica sites.*

Proof. The commit order at site S_i is indicated by updating the i-th index of the site version vector. This value is written into the redo-log entry for the transaction commit as a log sequence number (Chapter 2.2). Log entries from the site arrive at least once at other sites (Property 1), and if the replica site was unavailable, it retrieves the log entries from the persistent redo-log (Property 2). So, all log entries are guaranteed to arrive at least once at replica sites. A priority queue of per-site buffered log entries is maintained at each site, which de-duplicates and orders the entries by log sequence number (Chapter 2.2). A log entry per log sequence number is placed in the priority queue. Every transaction that updates data creates a single log entry containing the transaction's redo information for its refresh transaction, and all refresh transactions originating from a site are placed in the priority queue via the corresponding log entry. These log entries are applied serially in order as refresh transactions and the replica site is blocked from applying the next log entry in commit order until it arrives. Hence if T_1 commits before T_2, then T_1's commit timestamp is smaller than T_2's. Thus T_1's log entry is applied as $R(T_1)$ before T_2's log entry as $R(T_2)$. Hence $R(T_1)$ and $R(T_2)$ are applied in commit order at replica sites. □

To show that DynaMast provides SI, I prove the following lemmas and theorem:

Lemma 3. *A transaction T_1 must see the updates made by a transaction T_2 that has a commit timestamp smaller than T_1's begin timestamp.*

Proof. As described in Chapter 3.2.1, when a transaction T_2 commits, it updates the site version vector. Hence if T_1 has a larger begin timestamp than T_2's commit timestamp, it must read T_2's update to the site version vector. Given such a begin timestamp, the MVCC protocol described in Chapter 3.4.1 guarantees that T_1 will read T_2's updates to records, as T_2's committed versioned records will have a version number smaller than or equal to T_1's begin timestamp. □

Theorem 1. *If two transactions T_1 and T_2 have overlapping begin and commit timestamps then T_1 and T_2 can commit only if T_1 and T_2 write different data items.*

Proof. The transaction version vectors capture the begin and commit timestamps of the transaction. Therefore, T_1 has begin and commit timestamps $tvv_{B(T_1)}[\,]$ and $tvv_{T_1}[\,]$ respectively. Similarly T_2 has begin and commit timestamps $tvv_{B(T_2)}[\,]$ and $tvv_{T_2}[\,]$ respectively.

Assume, per the theorem, that both transactions T_1 and T_2 write data items, that is neither transaction is a read. Therefore, from the algorithm description in Chapter 3.2.1 it is known that $tvv_{T_1}[\,]$ and $tvv_{T_2}[\,]$ are unique.

Recall from Chapter 3.2.1 that a transactions begin and commit timestamps differ only in the i-th position if site S_i is the site that executed the transaction. Therefore, I consider two cases: when T_1 and T_2 both execute at site S_i, and when T_1 and T_2 execute at sites S_1 and S_2 respectively, without loss of generality.

Case 1 In the first case, if both transactions execute at site S_i, then the underlying data site provides SI (Lemma 1). Thus T_1 and T_2 cannot update the same data item and have overlapping timestamps; so the theorem holds.

Case 2 I now show, by way of contradiction, that it is not possible for T_1 and T_2 to execute at different sites, update the same data item d, and have overlapping begin and commit timestamps. Suppose, without loss of generality that T_1's begin timestamp is before T_2's, that is $tvv_{B(T_1)}[\,] < tvv_{B(T_2)}[\,]$.[1] Then for T_1 and T_2 to overlap, $tvv_{T_1}[\,] \geq tvv_{B(T_2)}[\,]$ must hold.

If T_1 and T_2 both update the same data item d on sites S_1 and S_2 respectively, then the adaptation advisor remastered d from S_1 to S_2 as $tvv_{B(T_1)} < tvv_{B(T_2)}$. As described in Chapter 3.2.2, the `release` request at S_1 will not complete until transactions that update d complete (Property 3), and `grant` does not return until all refresh transactions to d complete at S_2. The adaptation advisor does not allow transactions that update d to begin while the remastering protocol executes. The update propagation algorithm guarantees that the replication manager will apply all aforementioned updates (Lemma 2). Therefore, before T_2 begins its transaction, S_2's site version vector reflects the update from T_1's commit, as well as a remastering operation that increments the site version vector. Thus $svv_2[\,] > tvv_{T_1}[\,]$.

However, the system sets T_2's begin timestamp $tvv_{B(T_2)}[\,]$ to at least $svv_2[\,]$, which is a contradiction because $tvv_{T_1}[\,] < svv_2[\,] \leq tvv_{B(T_2)}[\,]$ and $tvv_{T_1}[\,] \geq tvv_{B(T_2)}[\,]$ cannot both hold. Therefore these transactions must have disjoint write sets, which satisfies the theorem. □

Lemma 4. *There exists a total commit order between transactions T_1 and T_2.*

Proof. I show the total commit order for transactions T_1 and T_2, by considering four cases.

Case 1: For all i, $tvv_{T_1}[i] < tvv_{T_2}[i]$, thus T_1 commits before T_2.

Case 2: For all i, $tvv_{T_1}[i] = tvv_{T_2}[i]$, thus T_1 and T_2 have the same commit time, and hence are placed in the same position in the total order. Note that in this case, at most one of T_1 and T_2 are update transactions, as update transactions must update a site version vector, and hence the commit timestamp (Algorithm 2).

[1] I use the definition that a vector $v_1[\,]$ is less than another vector $v_2[\,]$, ($v_1[\,] < v_2[\,]$), if $v_1[i] < v_2[i]$ for all positions i.

Case 3: There exists sites S_1 and S_2, such that $tvv_{T_1}[1] < tvv_{T_2}[1]$ and $tvv_{T_1}[2] = tvv_{T_2}[2]$. Then T_1 commits before T_2.

Case 4: There exists sites S_1 and S_2, such that $tvv_{T_1}[1] < tvv_{T_2}[1]$ and $tvv_{T_2}[2] > tvv_{T_2}[1]$. Such a scenario arises from write skew, and as outlined in Chapter 3.4.1 the adaptation advisor determines the commit order. Observe that this situation occurs from updating data items d_1 and d_2 mastered at sites S_1 and S_2 concurrently. Transactions reading these data items can observe one of four states, neither of T_1 or T_2's updates, both of T_1 and T_2's updates, T_1's updates, but not T_2's, or T_2's updates both not T_1's. Recall that the adpatation advisor uses locking (Algorithm 2) to ensure that all transactions can observe the same three states, by eliminating one of: T_1's updates but not T_2's, or T_2's update but not T_1's. This elimination determines the commit order: if T_1's update can be observed but not T_2, then T_1 commits before T_2. Otherwise, T_2, commits before T_1.

By way of example consider $tvv_{T_1} = [1, 0]$ and $tvv_{T_2} = [0, 1]$. These transactions race to acquire the lock to update the adaptation advisor's view of the site version vector. Initially this value is $[0, 0]$. If T_1 wins the race to acquire the lock (Algorithm 2, Line 34), then the adaptation advisor will set the site version vector to be $[1, 0]$, and subsequently, T_2 will update it to be $[1, 1]$. Hence a total commit order of T_1 then T_2, as it is not possible to observe a site version vector state $[0, 1]$, which would indicate the anomalous state: T_2's update but not T_1's.

I have outlined all four possible cases given transaction commit timestamps and defined a commit order; hence there exists a total commit order. □

Lemmas 3 and 4 and Theorem 1 together provide the requirements of SI, therefore DynaMast guarantees SI.

A.1.2 Dynamic Mastering with Session Based Snapshot Isolation

Recall from Chapter 3.2.1 that in addition to providing snapshot isolation (SI), the system also provides strong-session snapshot isolation (SSSI) [56]. To provide SSSI, the system enforces a *freshness rule* that ensures that a client's transaction executes on data that is at least as fresh as the state last seen by the client. Importantly, this *freshness rule* protects clients from reading data that is older than what they last read. Generally, clients see data that is more up-to-date than what they last observed because the replication scheme does not unnecessarily delay update propagation.

Dynamic Mastering Strong Session Snapshot Isolation Level Proof

I now prove that the update propagation protocol together with the session-based freshness scheme enforce the ordering guarantees required to provide SSSI.

I first prove the session requirements of SSSI:

Theorem 2. *If two transactions T_1 and T_2 belong to the same session, and the commit of T_1 precedes the start of T_2 then T_2's begin timestamp is greater than T_1's commit timestamp, that is T_2 observes any state observed or created by T_1.*

Proof. In DynaMast, the begin timestamp of a transaction is at least as large as the last commit timestamp of the previous transaction in the session. Therefore, if $tvv_{B(T_2)}[\,]$ is the begin timestamp of T_2, $tvv_{T_1}[\,]$ is the commit timestamp of T_1, and $cvv[\,]$ is the client session vector, then $tvv_{T_1}[\,] = cvv[\,] \leq tvv_{B(T_1)}[\,]$.

The blocking rules described in Appendix A.1.2 guarantee that for any site of execution, T_2 will not begin the transaction until the sites version vector is at least $cvv[\,]$. Consequently, T_2 will execute and observe any state reflected in $cvv[\,]$, as required by SSSI. □

Given that DynaMast provides SI, a pre-requisite to providing SSSI, the session requirements of SSSI, and a total commit order, DynaMast guarantees SSSI.

A.2 MorphoSys Provides Strong Session Snapshot Isolation

I now prove that MorphoSys preserves strong-session snapshot-isolation (SSSI). I first prove that MorphoSys preserves snapshot-isolation (SI) without physical design change operators (Appendix A.2.1). Then I consider the effect that physical design change operators have on SI (Appendix A.2.2). Finally, I put these two aspects of the proof together to prove MorphoSys provides SSSI (Appendix A.2.3).

A.2.1 Snapshot Isolation

I now prove that MorphoSys provides SI in the absence of physical design changes.

Lemma 5. *A transaction T_1 observes the updates made by a transaction T_2 that has a commit timestamp smaller than T_1's begin timestamp.*

Proof. Let T_1^B be transaction T_1's begin timestamp and T_2^C be transaction T_2's commit timestamp. If $T_2^C \leq T_1^B$ then there exists a partition p such that $T_2^C(p) \leq T_1^B(p)$. Suppose that T_2 updated a data item d in partition p, then as stated in Chapter 4.3.2, T_2 creates a new version of d assigned value $T_2^C(p)$. Since there are no physical design changes, then when T_1 performs a read operation on data item d, it occurs in partition p. As described in Chapter 4.3.2, when T_1 performs a read operation on d, in partition p, T_1 reads the largest version of d, denoted as $v(d)$, such that $v(d) \leq T_1^B(p)$ holds. Given that $T_2^C(p) \leq T_1^B(p)$, and T_2 performed an update, then there must exist a $v(d)$ such that $v(d) = T_2^C(p)$. Thus, when T_1 performs its read $T_2^C(p) \leq v(d)$ must hold. Hence, T_1 must observe T_2's update, or some later update to the data item. □

Lemma 6. *If two transactions T_1 and T_2 have overlapping begin and commit timestamps, then T_1 and T_2 can commit only if T_1 and T_2 write different data items.*

Proof. I say that two transactions T_1 and T_2 have overlapping begin and commit timestamps, if there exists a partition p that was in both of T_1 and T_2's read or write sets, and the begin and commit version numbers for p (that is $T_1^B(p), T_1^C(p), T_2^B(p), T_2^C(p)$) overlap. Recall, from Chapter 4.3.2 that for read-only transactions T, $T^B = T^C$. Additionally, if a transaction T updates a partition p, then $T^B(p) + 1 = T^C(p)$, as an update transaction acquires a mutually exclusive partition lock. Using this information I now prove, by way of contradiction, that the lemma holds.

Assume both transactions are updates and update the same data item, otherwise the lemma is trivially false. Additionally, as no physical design changes occur, then d is in the same partition p. Then, $T^B(p) + 1 = T^C(p)$ for both T_1 and T_2. In the first case, $T_1^B(p) \leq T_2^B(p) < T_2^C(p) \leq T_1^C(p)$. In the second case, $T_1^B(p) \leq T_2^B(p) < T_1^C(p) \leq T_2^C(p)$.

Observe that in both cases, for both of these inequalities to be true, and for $T^B(p)+1 = T^C(p)$ to hold for both T_1 and T_2, then it must be that $T_1^B(p) = T_2^B(p)$ and $T_1^C(p) = T_2^C(p)$. However, as stated in Chapter 4.3.2 transactions acquire partition locks before reading the partition version number and constructing T_B and release the locks after setting T_C and updating the partition version number. As I assume no physical design changes, then T_1 and T_2 must execute at the same site. Thus one of T_1 or T_2 will acquire the partition lock first, so $T_1^B(p) \neq T_2^B(p)$ must hold. However this inequality is a contradiction with $T_1^B(p) = T_2^B(p)$ □

Lemmas 5 and 6 satisfy the requirements of SI, under the assumption of the correctness of the begin and commit timestamps. I now prove that the begin and commit timestamps produce a consistent snapshot of data items across partitions.

Lemma 7. *MorphoSys' transactions begin and commit timestamps produce snapshot consistent state. That is, if T_2 updates data items in partitions p_1 and p_2, and T_1 reads data items in partitions p_1 and p_2, then either $T_2^C \leq T_1^B$, and hence T_1 will observe T_2's updates, or $T_2^C > T_1^B$ and T_1 will not observe T_2's updates.*

Proof. Observe that for this proof to hold, either $T_2^C(p) \leq T_1^B(p)$ must hold for both p_1 and p_2, in which case it follows from Lemma 5 that T_1 observes T_2's updates, or, $T_2^C(p) > T_1^B(p)$ must hold for both p_1 and p_2, in which case T_1 will not observe T_2's updates.

Recall, from Chapter 4.3.2, that if T_2 updates partitions p_1 and p_2, then the update results in a *depends* relationship between p_1 and p_2. That is, if $T_2^C(p_1)$ and $T_2^C(p_2)$ are the versions of p_1 and p_2 updated by T_2, then MorphoSys records $T_2^C(p_2) = \textit{depends}(p_1, T_2^C(p_1), p_2)$ and $T_2^C(p_1) = \textit{depends}(p_2, T_2^C(p_2), p_1)$. It follows that there are three cases in the construction of T_1's begin timestamp.

Case 1: Suppose that T_1 initially selects $T_1^B(p)$ such that $T_2^C(p) \leq T_1^B(p)$ for both p_1 and p_2. This case trivially results in $T_2^C \leq T_1^B$, hence T_1 observes both of T_2's updates.

Case 2: Suppose that T_1 initially selects $T_1^B(p)$ such that $T_2^C(p) > T_1^B(p)$ for both p_1 and p_2. This case trivially results in $T_2^C > T_1^B$, hence T_1 does not observe either of T_2's updates.

Case 3: Suppose, without loss of generality, that T_1 initially selects $T_1^B(p_1)$ and $T_1^B(p_2)$, such that $T_2^C(p_1) > T_1^B(p_1)$ and $T_2^C(p_2) \leq T_1^B(p_2)$ holds. Following the consistent read rules, T_1 updates $T_1^B(p_i)$ such that $T_1^B(p_i) \geq \textit{depends}(p_j, T_1^B(p_j), p_j)$ for all p_i and p_j in its read set. Setting p_i as p_1 and p_j as p_2, then $T_1^B(p_1) \geq \textit{depends}(p_2, T_1^B(p_2), p_1)$. As $T_1^B(p_2) \geq T_2^C(p_2)$, and the depends relationship always uses the max operator (Equation 4.1), then $\textit{depends}(p_2, T_1^B(p_2), p_1) \geq \textit{depends}(p_2, T_2^C(p_2), p_1)$. Recall that $\textit{depends}(p_2, T_2^C(p_2), p_1) = T_2^C(p_1)$, so combining all equations $T_1^B(p_1) \geq T_2^C(p_1)$. Thus, $T_1^B(p) \geq T_2^C(p)$ for both p_1 and p_2, a contradiction as I selected $T_2^C(p_1) > T_1^B(p_1)$. Hence, $T_2^C \leq T_1^B$, and T_1 observes both of T_2's updates.

□

Lemma 8. *There exists a total commit order between transactions T_1 and T_2.*

Proof. The proof follows from Lemma 4 but replaces site version vector's with per-partition version information.

Case 1: For all p, $T_1^C(p) < T_2^C(p)$, thus T_1 commits before T_2.

Case 2: For all p, $T_1^C(p) = T_2^C(p)$, thus T_1 and T_2 have the same commit time, and hence are placed in the same position in the total order. Note that in this case, at most one of T_1 and T_2 are update transactions.

Case 3: There exists p_1 and p_2, such that $T_1^C(p_1) < T_2^C(p_1)$ and $T_1^C(p_2) = T_2^C(p_2)$. Then T_1 commits before T_2.

Case 4: There exists p_1 and p_2, such that $T_1^C(p_1) < T_2^C(p_1)$ and $T_1^C(p_2) > T_2^C(p_2)$. Such a scenario arises from write skew, and as outlined in Chapter 4.3.2 the adaptation advisor determines the commit order. Transactions reading partitions p_1 and p_2 can observe one of four states, neither of T_1 or T_2's updates, both of T_1 and T_2's updates, T_1's updates, but not T_2', or T_2's updates both not T_1's. Recall that the adpatation advisor uses locking (Algorithm 3) to ensure that all transactions can observe the same three states, by eliminating one of: T_1's updates but not T_2's, or T_2's update but not T_1's. This elimination determines the commit order: if T_1's update can be observed but not T_2, then T_1 commits before T_2. Otherwise, T_2, commits before T_1.

I have outlined all four possible cases given transaction commit timestamps and defined a commit order; hence there exists a total commit order. □

Together, Lemmas 5, 6, 7 and 8 prove that MorphoSys satisfies the requirements of SI, when there are no physical design changes.

A.2.2 Physical Design Changes and Snapshot Isolation

Appendix A.2.1 proved that MorphoSys provides SI when there are no physical design change operators. I now prove that MorphoSys provides SI in the presence of these physical design change operators. I make four critical observations about the proofs in Appendix A.2.1, and the changes that arise in the presence of physical design changes.

First, removing a replica does not change the correctness of the proofs, as a replica partition is no longer present at a data site.

Second, adding a replica does not change the correctness of the proofs. Recall from Chapter 4.3.4, that MorphoSys installs a snapshot of the replicated partition that includes the partition version number, the *depends* relationship, and versioned data items. Hence, as replicas execute only read-only transactions, all of the state necessary to ensure the correctness of Lemmas 5 and 7 exist at the newly created replica, or in the case of Lemma 8, at the adaptation advisor.

Third, to prove Lemma 6, I assumed that updates to partitions occurred on the same master data site. This assumption does not hold if the system remasters a partition. Hence, I need only prove that a partition cannot be updated at two data sites concurrently, as a consequence of remastering. Recall, from Chapter 4.3.4, that remastering occurs transactionally, and hence the *remaster* operator acquires the partition lock. Additionally, after the old master site releases the mastership of a partition, it can no longer service update transactions to the partition. Furthermore, the new master site does not become the new master until it applies the propagated update releasing the mastership from the old master and all previous updates to the partition. Consequently, no update transactions to the partition occur while it is being remastered. Thus, given two transactions that update the same data item (and thus partition), either the updates both occur at the same site, proven correct in Lemma 6, or one transaction (T_1) updates the data item at the old master, and the other (T_2) at the new master. However, because the new master blocks updates until all previous updates to the partition are applied, then $T_1^B(p) < T_1^C(p) \leq T_2^B(p) < T_2^C(p)$, hence the updates do not overlap.

Fourth, in Lemmas 5 and 6 I assumed that data items always belong to the same partition. This assumption does not hold if the system splits partitions apart, or merges them together. Recall from Chapter 4.3.4, that MorphoSys performs these operations while holding the partitions' locks, consequently, after the *split* or *merge* operator completes subsequent transactions execute on the newly created partitions. Additionally, these physical design change operators assign the newly created partition's version numbers as the maximum of the original partitions' version number and induce a *depends* relationship among the partitions. Thus, as shown in the proof in Lemma 7, subsequent transactions generate transaction begin timestamps that observe snapshot consistent state. Consequently, the dependency relationship determines the transactions total commit order, as the transaction occurs either before or after the *split* or *merge* operation.

Given the fourth observation, I must prove that updates to the same data item do not occur concurrently, in the presence of *split* or *merge* operations. Similar to the remastering case, updates to the same data item either occur in the same partition, or, before and after a *split* or *merge* operation, and thus in a different partition. In the former case, Lemma 6 holds. Considering the latter case, without loss of generality, suppose T_1 updates data item d in partition p, p is *split* into p_L and p_H, and T_2 updates d that is now contained in p_L. By definition, $T_1^B(p) < T_1^C(p)$ and $T_2^B(p_L) < T_2^C(p_L)$. By construction of p_L, the intial partition version of p_L, $v(p_L) \geq T_1^C(p)$, and T, hence $T_1^C(p) \leq T_2^B(p_L) < T_2^C(p_L)$. Additionally, as p does not exist for T_2, but is stored as part of the *depends* relationship, thus $T_1^C(p) \leq \mathit{depends}(p_L, T_1^B(p_L), p) = T_2^B(p)$. Hence, $T_1^B(p) < T_1^C(p) \leq T_2^B(p) = T_2^C(p)$ and thus the updates do not occur concurrently. A similar argument follows for the *merge* operator.

Following from the four observations and associated proofs, MorphoSys provides SI in the presence of physical design changes.

A.2.3 Enforcing Strong Session Snapshot Isolation

I now prove that MorphoSys provides SSSI by proving the session requirement of SSSI.

Theorem 3. *If two transactions T_1 and T_2 belong to the same session, and the commit of T_1 precedes the start of T_2, then T_2's begin timestamp is greater than T_1's commit timestamp.*

Proof. Recall from Chapter 4.3.2 and Chapter 4.3.2, that MorphoSys tracks a session timestamp C^S, composed of the maximum observed $T^C(p)$ for all transactions T in the same session, and accessed partitions p. Thus $T_1^C \leq C^S$. Furthermore, MorphoSys uses this session timestamp as the initial transaction begin timestamp, before updating it based on observed partition version numbers, blocking if necessary. Thus $C^S(p) \leq T_2^B(p)$. Combining the two inequalities, $T_1^C \leq T_2^B$, as required. □

Theorem 3, together with Lemma 5, prove that if T_1 and T_2 belong to the same session, and the commit of T_1 precedes the start of T_2, then T_2 observes any state observed or created by T_1. Given that MorphoSys guarantees the session requirements of SSSI in addition to SI, MorphoSys thus guarantees SSSI.