ABSTRACT

Title of Proposal: HIGH PERFORMANCE DISTRIBUTED TRANSACTIONS FOR MULTI-REGION DATABASE SYSTEMS

Cuong D. T. Nguyen
Doctor of Philosophy, 2024

Dissertation Directed by: Professor Daniel J. Abadi
Department of Computer Science

The growing popularity of global applications, such as social networks, online gaming, and supply chain management, has spurred the demand for a more advanced database system layer in multi-region cloud infrastructures. This layer should provide a strongly consistent interface, abstracting the complexity of distributed computing so developers can focus on application logic without worrying about data race issues. At the same time, it must leverage geographic redundancy to tolerate failures, ranging from single-node crashes to regional outages. Achieving these goals often conflicts with the need for scalability and high performance. Ensuring strong data consistency requires additional coordination between nodes, while enhancing availability and durability requires data replication. These measures increase latency and resource consumption, thereby limiting performance and scalability. As a result, existing solutions, such as NoSQL database systems, often compromise with weaker consistency levels, while fully geo-replicated systems incur performance and scalability penalties to achieve stronger consistency guarantees.

This dissertation addresses these challenges in three directions. First, we take a retrospective approach by enhancing an existing production-ready shared-storage architecture, derived from the Amazon Aurora database system. Our proposed

modifications enables Aurora-style systems to support multiple writer nodes across geographically dispersed regions. Evaluation of our implementation, SunStorm, demonstrates substantial reductions in latency and improved scalability. Second, we adopt a forward-looking perspective by investigating a more recently proposed architecture known as deterministic database systems. We develop a new protocol within this architecture that facilitates the processing of strictly serializable multi-region transactions. Our implementation, Detock, achieves minimal performance degradation under high-contention workloads, improving throughput by an order of magnitude compared to state-of-the-art approaches and reducing latency by up to a factor of five. Finally, we conduct a comprehensive empirical study across a diverse range of real-world applications—such as e-commerce, chat, blogs, and content management systems—reassessing the assumptions of recent database system proposals and providing valuable insights for future transactional database research.

HIGH PERFORMANCE DISTRIBUTED TRANSACTIONS FOR MULTI-REGION DATABASE SYSTEMS

by

Cuong D. T. Nguyen

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2024

Advisory Committee:
Professor Daniel J. Abadi, Chair/Advisor
Professor Louiqa Raschid, Dean's Representative
Professor Amol Deshpande
Professor Pete Keleher
Professor Alan Zaoxing Liu

Acknowledgements

I am deeply grateful to my advisor, Daniel Abadi, for his support and guidance throughout my PhD journey. Dan consistently believed in me and gave me the space to explore my own ideas and methods, while always being available whenever I sought his advice. His succinct explanations clarified my thoughts and shaped my understanding of many complex concepts in my field of study. His pursuit of simplicity and practicality left a lasting impact on my way of thinking, which I believe has made me a better researcher.

I extend my thanks to Prof. Louiqa Raschid, Prof. Amol Deshpande, Prof. Pete Keleher, and Prof. Alan Zaoxing Liu for their time and effort in serving on my dissertation committee. Their valuable insights and constructive suggestions greatly enhanced the quality of this work.

I also appreciate the opportunities to collaborate with senior researchers who broadened my perspective on database system research. My sincere thanks go to Dr. Yinan Li, who mentored me during my internship at Microsoft Research, and to Prof. Xiangyao Yu at the University of Wisconsin, Madison, who provided insightful feedback in the research project we collaborated on.

I feel fortunate to have worked with many wonderful collaborators, all of whom I would like to thank—Pooja Nilangekar, Heikki Linnakangas, Johann Miller, Kevin Niechen, Chris DeCarolis, Zhenghong Yu, and Zhihan Guo. In particular, Pooja was the colleague, beside Dan, with whom I had the longest and most fruitful discussions

on transaction processing. Heikki's expertise on the PostgreSQL codebase provided invaluable insights that helped us overcome several challenging roadblocks.

I dedicate this thesis to my family. My parents have been with me every step of the way, offering both physical and emotional support. My mom expressed her love through the daily comfort of home-cooked meals, while my dad always listened attentively, showing genuine interest in my research as I tried to explain it in simple terms. They never missed a moment to notice when I was struggling and provided the encouragement I needed to keep going. I also want to thank my sister and brother-in-law, who took care of me during my time in Vietnam while working on this thesis. Every small act of love and support from my family carried me across the finish line.

I would like to express my heartfelt gratitude to my dear wife, Phụng, for her unconditional love and unwavering support. She is a ball of energy whose positivity has brightened up even my darkest days. I owe my deepest thanks to her patience and dedication, as she stayed by my side and cared for me throughout this long and challenging journey.

I am grateful to my childhood teachers, cô Lệ, who laid the foundation for my programming knowledge and ignited my passion for the field, and thầy Tuấn, who guided me through my early academic achievements. I also wish to thank Prof. Rodica Neamtu, whose influence inspired me and set me on the path to pursuing this PhD.

I want to thank my Vietnamese friends that I met in Maryland: anh Khánh & chị Huyền, anh Khôi & chị Uyên, anh Phong & Khoa, anh Tuấn, anh Trí, anh Huy, Khoa & Thủy, Chương & Quỳnh, Nhật, Tín, Phương, and Đăng. I forever cherish the wonderful memories we created together.

Finally, I want to thank the "SEA gang" of SIGMOD 2023—Wan Shen Lim and Bobbi Yogatama—for making my first major conference experience so much more joyful and memorable.

Table of contents

List of Figures

Chapter 1

Introduction

Over the past decade, the deployment of database management systems (DBMS) has undergone a significant shift from on-premise to cloud environments. According to a 2022 market report by Gartner [112], cloud-based DBMS spending has surpassed that of on-premise solutions, with cloud accounting for 55.2% and on-premise accounting for 44.8%. The widespread adoption of cloud infrastructure, which provides developers with easy access to worldwide networks of data centers, has lowered the barrier to entry for applications to become global. These applications—ranging from social networks and e-commerce to logistics and security—promise low-latency user experiences and high availability by leveraging deployments in multiple geographically dispersed regions, thereby bringing data closer to users and enhancing redundancy.

However, the evolving requirements of these applications demand a more advanced database system layer. This layer should provide a strongly consistent interface, abstracting the complexity of distributed computing so developers can focus on application logic without worrying about data race issues. At the same time, it must leverage geographic redundancy to tolerate failures, ranging from single-node crashes to regional outages. Achieving these goals often conflicts with the need for scalability and high performance. Ensuring strong data consistency requires additional

coordination between nodes, while enhancing availability and durability requires data replication. These measures increase latency and resource consumption, thereby limiting performance and scalability.

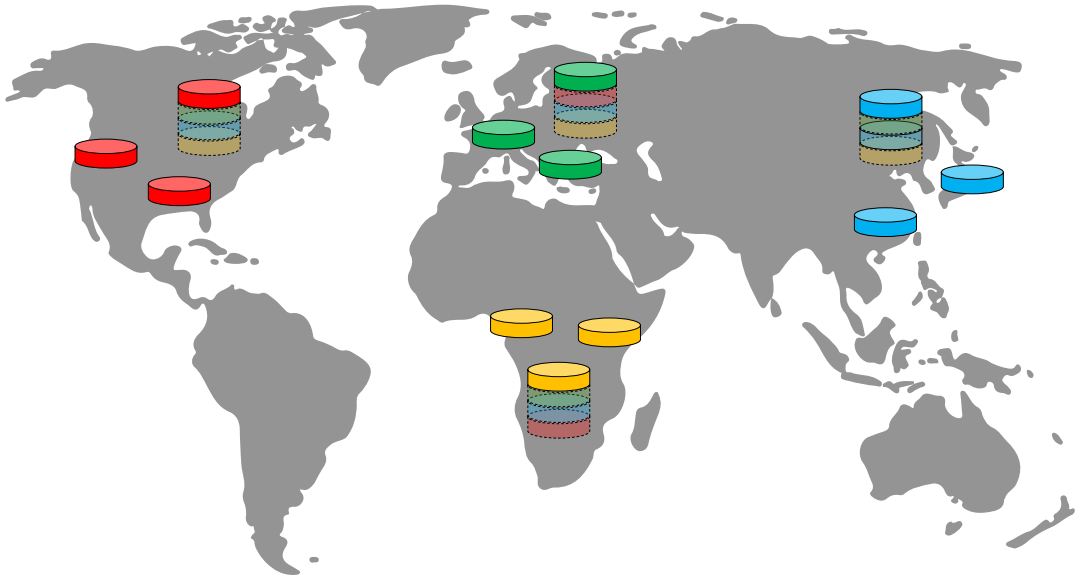


Figure 1.1: Geographic partitioning among four regions

These challenges are further exacerbated when serving data from remote regions, as cross-region network latency can reach hundreds of milliseconds [133]. Unfortunately, this issue is difficult to eliminate completely due to the physical limitation imposed by the speed of light. Even the best hypothetical link through the Earth's core requires a minimum round-trip time of 85.1ms [9]. Nevertheless, for workloads with a high degree of *locality*, where many subsets of data items are often accessed together, **geographic partitioning** (or geo-partitioning) can be utilized to minimize cross-region communications. In this scheme, data is split into partitions each of which is placed in the region where it is most frequently accessed. For instance, Fig. 1.1 illustrates a database divided into four partitions—red, green, yellow, and blue—located in North America, Europe, Africa, and East Asia, respectively. Each partition can be additionally replicated within its local region for increased availability and

durability without significantly compromising performance. Furthermore, a region can host replicas of remote partitions (depicted by translucent blocks in Fig. 1.1) to reduce latency of remote reads. These remote partitions are often *asynchronously* replicated to avoid performance penalties.

Traditional database systems, such as PostgreSQL and MySQL, offer ACID transactions with consistency and isolation at the highest level of *strict serializability* [94]. However, these systems were designed to run on a single node with limited fault-tolerance capability. Therefore, they struggle to meet the requirements for scalability and high availability over multiple nodes or regions. In contrast, NoSQL database systems like MongoDB and Apache Cassandra address scalability, performance, and availability issues but compromise on strong consistency and offer limited transactional support, exposing applications to hard-to-detect bugs and security vulnerabilities [132]. Recently, there emerge new distributed SQL databases that are based on the architecture of Google Spanner [31], such as YugabyteDB [144] and CockroachDB [118], [128]. These systems promise high availability, scalability, serializable ACID transactions, and geo-partitioning, all while maintaining convenient SQL interfaces. However, as discussed in Chapter 3 and 4, their extensive use of transaction coordination can lead to performance issues, particularly in geo-partitioned settings.

This dissertation tackles the aforementioned challenges in three directions. Firstly, we adopt a retrospective approach by adding geo-partitioning capabilities to an existing shared-storage architecture, which is based on traditional DBMS and originates from Amazon Aurora [129]. Secondly, we adopt a forward-looking perspective by exploring recent advancements in transactional systems, specifically those among *deterministic database systems* [2], and develop a novel protocol to mitigate high-latency issues in cross-region transactions of a deterministic geo-partitioned system. Finally, we take a comprehensive view by conducting an empirical study of diverse real-world applications, reassessing assumptions of recent database system proposals and providing valuable

insights to guide future research in transactional database systems.

Improving existing Amazon-style shared-storage systems. (Chapter 3) There are two main approaches to scaling transactional database workloads for cloud DBMS: (1) a shared-nothing architecture with distributed transaction processing and commit protocols, or (2) an Aurora-style shared-storage architecture with separate compute and storage layers that scale independently. In option (2), the compute layer typically contains a single writer node and all other compute nodes have read-only access to the data. This may lead to scalability limits for write-intensive workloads relative to the shared-nothing approach that can divide write effort across multiple nodes. Furthermore, it introduces communication latency for write transactions that initiate far from the writer node. However, shared-nothing systems must pay the overhead of distributed coordination and commit protocols during transaction processing. We propose the design of a more scalable version of Aurora-style systems which supports multiple writer nodes managing geographically partitioned data. It yields many of the efficiency benefits of Aurora-style systems while removing the scalability bottleneck. Furthermore, the use of geographic partitioning improves latency by over an order of magnitude for global applications in which clients from across the world can experience local write performance instead of having to communicate with a single primary server.

New deterministic concurrency control protocol. (Chapter 4) Many globally distributed DBMS need to replicate data across large geographic distances. Since synchronously replicating data across such distances is slow, those systems with high consistency requirements often geo-partition data and direct all linearizable requests to the primary region of the accessed data. This significantly improves performance for workloads where most transactions access data close to where they originate from. However, supporting serializable multi-geo-partition transactions is a challenge, and they often degrade the performance of the whole system. This becomes even

more challenging when they conflict with single-partition requests, where optimistic protocols lead to high numbers of aborts, and pessimistic protocols lead to high numbers of distributed deadlocks. We describe the design of concurrency control and deadlock resolution protocols, built within a practical, complete implementation of a geo-distributed database system called DETOCK, that enables processing strictly serializable multi-region transactions with near-zero performance degradation at extremely high conflict and order of magnitude higher throughput relative to state-of-the-art approaches, while improving latency by up to a factor of 5. DETOCK is part of a long line of research on deterministic database systems, which are discussed in detail in Section 2.3.

Study on assumptions of transactional database systems. (Chapter 5) Database systems often provide an interactive interface to execute multiple command round-trips within a single transaction. However, the evolution of application workloads has prompted a shift toward transactional systems that either completely eliminate interactive transactions or optimize for transactions with minimal round-trips. Additionally, some systems rely on upfront knowledge of the transactions' read/write sets to employ techniques that enhance the performance of concurrency control. However, the rationales for giving up transaction interactivity and assuming read/write set knowledge have not been empirically validated. To address this, we conduct an extensive study of 108 open-source applications, analyzing over 30,000 transactions to evaluate the prevalence of these design assumptions. We created a dataset by annotating all database operations in each application according to definitions related to read/write set inferability and transaction interactivity, and then computed statistics from this dataset. Our study reveals that for 90 of applications, at least 58% of transactions have read/write sets that can be inferred in advance. Transactions that cannot be inferred may need a fallback mechanism, which could be designed to be lightweight. Furthermore, our findings indicate that DBMS without

interactive transactions can support 60% of applications without modification, and most interactive transactions can be converted into one-shot transactions with minimal changes. These insights underscore the potential for further optimization and research in designing DBMS that balance transaction expressivity and performance.

Chapter 2

Background

This chapter provides the necessary background on geo-distributed database systems and the architectures considered in this thesis. Specifically, the SunStorm system in Chapter 3 is built upon an architecture introduced by Amazon Aurora and the DETOCK system in Chapter 4 is part of a long line of research on deterministic database systems.

2.1 Geo-distributed database systems

As IT infrastructures increasingly transition to the cloud, more applications are being designed to serve users across the globe. A prime example of such applications is social networks, which aim to connect people regardless of geographic boundaries. Similarly, online multiplayer games have seen significant growth, offering real-time interactions between players worldwide. Other prominent examples include e-commerce platforms, which enable global trade and transactions and supply chain management systems that coordinate logistics across continents. For these applications, user experience is critical, and two of the most crucial performance metrics are latency and availability. Low latency ensures that users experience minimal delays in their interactions, while high availability guarantees that services are accessible whenever users need them.

Poor performance in either area can result in diminished user satisfaction, reduced retention rates, and ultimately a loss in profitability for businesses [20], [113].

To address these performance concerns, geo-distributed database systems have become an essential component in modern cloud infrastructures. These systems distribute data across multiple geographically dispersed locations, improving latency and availability. A key feature of geo-distributed systems is data replication, where data is replicated across several geographically distant nodes to ensure resilience and fault tolerance. In the event of a failure at one location, another location can continue operations with minimal disruption, thus maintaining system availability. Moreover, geo-replication improves read latency since read operations can be executed on data replicas closer to the where the operations originate from. However, achieving optimal performance in geo-distributed systems requires navigating the fundamental trade-offs between consistency, latency, and availability, formalized by the CAP theorem [18] and expanded by the PACELC theorem [1]. To cope with these trade-offs, many systems opt for weaker consistency models like eventual consistency [19], [35], [70], [73], [85], [97], [121] to reduce latency at the cost of increased application implementation complexity. On the other hand, there is also a large body of research focusing on maintaining strong consistency while mitigating latency penalties by reducing the number of cross-region communication round trips [49], [67], [82], [138], [149].

Recently, the geo-partitioning strategy introduced in Chapter 1 has gained attention in the literature [26], [109], [118], [128]. This strategy can be combined with geo-replication (partial geo-replication) to significantly improve latency for workloads with a high degree of locality, by avoiding cross-region communication for the majority of the workload. Chapter 3 and 4 extend the research on geo-distributed database systems with new techniques designed specifically for the geo-partitioning strategy.

2.2 Aurora-style database systems

In recent years, the architecture employed by Amazon Aurora [129], which decouples compute from storage, has gained significant popularity among cloud vendors [151]. This approach has been adopted by other database systems such as Google AlloyDB [89], Microsoft Socrates [6], PolarDB [23], [140], [141], and Neon [90]. While these systems differ in certain details and implementations, they share similar foundational design principles. Throughout this dissertation, we refer to this architecture as the **single-primary** architecture.

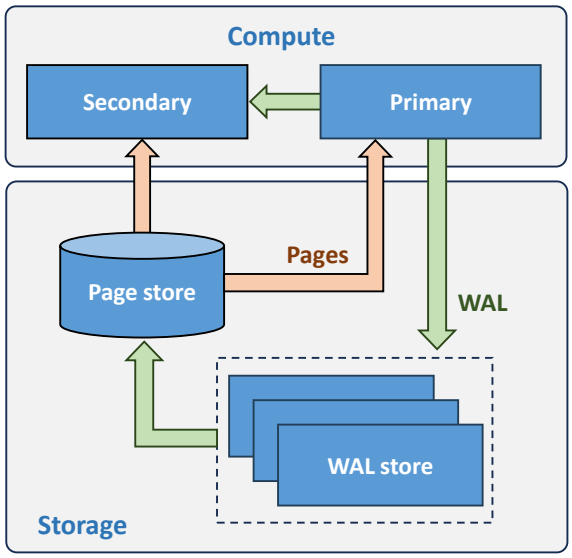


Figure 2.1: Single-primary architecture

Fig. 2.1 depicts the compute and storage layers of this architecture. The compute layer has a primary node, and optional secondary nodes, running a relational DBMS like PostgreSQL or MySQL. All transactions involving writes must be sent to the primary node, while secondary nodes only serve read-only transactions. Every data modification triggers the primary compute node to generate a **Write-Ahead Log** (WAL) record containing information for redoing the change on a specific page. The position of a record in the log is its **Log Sequence Numbers** (LSN).

Instead of reading from or writing to local disks, compute nodes forward these

requests to the storage layer. The primary compute node sends the WAL to a WAL store within the storage layer, comprising multiple log storage nodes. Together with the compute node, these log storage nodes participate in a consensus or quorum-based protocols to replicate WAL records, ensuring durability and fault-tolerance. Additionally, the WAL store forwards WAL records to the page store, which can reconstruct any page at a specified LSN.

Read requests from any compute node—whether primary or secondary—are directed to the page store with a page identifier and an LSN up to which all relevant WAL records must be applied to that page. Pages retrieved from the page store are cached in the compute node's DBMS and subject to its buffer pool management system. The primary compute node is the only writer, and thus is always aware of the most recently persisted WAL record. Secondary nodes receive log records either directly from the primary node (e.g., Aurora, Socrates) or from the WAL store (e.g., Neon) to keep their buffer pool pages up-to-date.

This architecture offers several advantages. By decoupling compute from storage, the two layers can scale independently and use hardware that suits their requirements. The storage layer can be transparently sharded to eliminate I/O bottlenecks. Failover time is reduced and more predictable since a new compute node can join and catch up quickly without copying data from another node. Data backup and archiving are easier as they can be done at the storage layer without interfering with the compute nodes' operation.

While this architecture can expand to multiple geographic regions to improve the performance for read-only transactions, all write transactions must go to the single region that hosts the primary node, resulting in increased latency of these transactions and a potential bottleneck at the primary region. Chapter 3 proposes an extension to this architecture that addresses these issues.

2.3 Deterministic database systems

Traditional DBMS are inherently non-deterministic. This non-determinism stems from factors such as operating system (OS) scheduling decisions, network behavior, and random number generation. While processing multiple concurrent transactions, these systems ensure serializability, meaning the final state of the database is equivalent to *some* serial execution. However, due to the non-deterministic factors, traditional DBMS do not guarantee which specific serial order the transactions will follow.

In contrast, deterministic database systems guarantee that a given initial state and set of input operations will always lead to a *single* final state. To achieve this, these systems rely on certain design principles, laid out in detail by Abadi et al. [2], which we summarize the most important points in the subsequent paragraphs.

The first design principle of deterministic database systems is input processing. Traditional database systems typically employ different communication threads to receive transactions from different clients and execute them independently. However, determinism requires a universally agreed-upon input among the participants (i.e., threads and nodes). Therefore, the system must employ some method to produce this canonical instance of the input. This can be as simple as routing all transactions through a single thread that orders them in a persisted log to using a consensus protocol such as Paxos [71], [72] or Raft [92] to form a replicated log over a cluster of nodes. This input processing step also involves evaluating and replacing non-deterministic code—such as getting current time or generating random numbers—inside transaction logic with the corresponding result.

The second design principle is that the OS's thread scheduling must not impact the database's state. For optimal performance, database transactions run concurrently in separate threads or processes. However, since the OS schedules threads in a fundamentally non-deterministic way, traditional databases face issues with race conditions that alter the database state. Instead, deterministic databases have their

own schedulers, operating at a higher level than the OS's thread scheduler. This approach allows for more efficient processing, as the lower-level scheduling layer cannot exploit application-specific semantics and would impose unnecessary restrictions and overhead to maintain determinism.

Finally, the commit status of transactions in deterministic database systems must not be affected by failures such as crashes of the OS or server hardware, which are fundamentally non-deterministic events. Therefore, deterministic databases typically restore their state to how it was right before the error and continue ongoing transactions in the input log from there, instead of aborting and restarting them upon recovery.

An example implementation that achieves determinism is Calvin [123]. To create the input log, nodes in Calvin collect transaction requests from the client and put them in batches every 10 milliseconds. The batches are then replicated with Paxos and distributed to every partition, which independently pieces together its own view of the input log in a deterministic, round-robin manner. Calvin schedules transactions using *ordered locking*, which mandates that locks are requested in the same order as transactions appear in the input log. Specifically, all locks for transaction X must be requested before any locks for transaction Y if X precedes Y in the log. Additionally, locks must be granted in the order they are requested. By adhering to this ordering, deadlocks are prevented, and the final database state after concurrent transaction execution is equivalent to the serial execution following the sole order of the input log, ensuring both determinism and strict serializability.

Deterministic execution offers several benefits. The most apparent benefit is in database replication. Since transactions are executed deterministically over a single input log, all replicas receiving this log are guaranteed to remain consistent. Unlike traditional DBMS where replication happens after transaction writes and potentially before their commits—extending the window during which conflicting transactions cannot run—deterministic database systems coordinate replication *prior to* transaction

execution. This pre-execution coordination occurs outside the conflict window, thereby not lengthening its duration. Since the first proposal for determinism in database systems by Thomson et al. [122], subsequent studies have uncovered many other benefits, including more efficient concurrency control and increased scalability [46]–[48], [56], [79], [91], [109], [110], [123]. For example, when multiple nodes are involved in committing a transaction, traditional systems use commit protocols such as 2-phase commit (2PC) to guard against node failures and ensure transaction atomicity. In contrast, deterministic database systems such as Calvin, by design, do not allow non-deterministic failures to affect commit statuses of transactions. Therefore, they can eliminate 2PC completely, significantly reducing commit protocol latency and increasing scalability and concurrency, especially under high-contention workloads [54], [111]. Other examples include BOHM [46], which shows that determinism can help efficiently schedule transactions in a system that uses multi-version storage, and PWV [47], which introduces a technique to reveal transactions' writes before the end of their execution, thereby increasing concurrency.

Deterministic database systems have some downsides. First, the input processing step may introduce extra latency to all transactions. However, this latency is often offset (and even surpassed) by the latency reduction techniques of deterministic systems, which shorten the duration of the commit protocol and can even commit transactions after partial execution [47]. In Chapter 4, we describe a method to address this latency issue in a multi-region environment. Second, deterministic database generally enable concurrency at the cost of requiring more information about transactions before their execution, particularly the set of data items that they will access and oftentimes the complete transaction code. This means that interactive transactions—where a client communicate with the server over multiple round-trips and thus does not send the complete transaction—are not well-supported. We revisit this information versus performance trade-off in Chapter 5 within the context of real-world applications.

Chapter 3

SunStorm: Geographically distributed transactions over Aurora-style systems

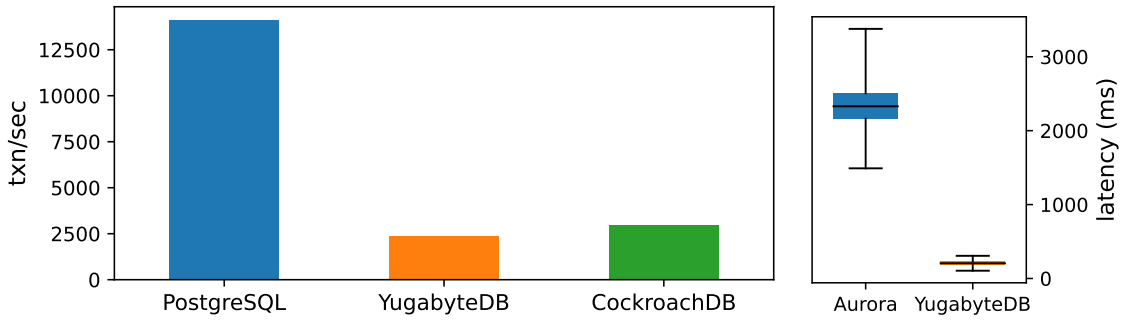
There are two major competing schools of thought on scaling transactional processing to tens of thousands (or more) of concurrent transactions. The first school of thought is to partition the database across a shared-nothing cluster of machines in one or more geographic regions. Every machine may contribute to transaction processing that accesses data within its partition. The larger the required scale, the more machines are added to the cluster to share load and meet these scale requirements. Examples of such systems include Calvin [123], Spanner [31], TAPIR [149], Carousel [138], CockroachDB [118], [128], YugabyteDB [144], OceanBase [142], Chardonnay [42], and Detock [91].

The other school of thought is to use a shared-storage architecture in which compute and storage are decoupled. Multiple compute nodes exist to handle the client workload, accessing the same shared storage layer (which typically consists of data partitioned and replicated across a cluster of servers, potentially also replicated across geographic regions). This design in which compute nodes have minimal long-term

state enables extremely high elasticity as new compute nodes can be added or dropped on the fly, without having to copy or move data from other nodes. However, by avoiding state, these compute nodes cannot support concurrency control protocols across multiple nodes. Therefore, they typically designate only a single compute node to support requests that update the data in the storage, and all other compute nodes only support read-access. Examples of such systems are Amazon Aurora [129], Google AlloyDB [89], Microsoft Socrates [6], PolarDB [22], [23], [140], [141], and Neon [90].

The first school of thought is fundamentally more scalable since it does not limit write access to a single instance [151]. However, it struggles with multi-partition transactions, which are notoriously challenging to perform at high levels of performance and correctness, and introduce much complexity into the system. These transactions typically require advanced distributed transaction functionality, which may include distributed concurrency control, coordination, commit, deadlock resolution, and consensus protocols which can increase the amount of code run per transaction by over an order of magnitude.

To illustrate the per-transaction overhead of the shared-nothing approach, Fig. 3.1a compares the throughput of PostgreSQL running on a single node with two state-of-the-art shared-nothing DBMSs running on a three-node cluster: CockroachDB [118] and YugabyteDB [144]. All systems ran on AWS EC2 c5d.4xlarge instances with



(a) Throughput (b) Latency

Figure 3.1: Single-writer vs. shared-nothing DBMSs

recommended configurations. We sent transactions with 8 read-modify-write operations to each system. PostgreSQL performs over 4 times faster than the distributed systems, despite having 1/3 the computing resources (and is thus more than 12 times more efficient on a per-machine basis). These results are not surprising and are well-known in the industry. Distributed transactions are complex. For a workload that can run easily on a single machine, it is most efficient to run it there and avoid the coordination required for correct, ACID-compliant distributed execution.

This phenomenon has driven the success of the aforementioned Aurora-style systems that are built according to the second school of thought and therefore avoid much of the complexity of distributed transactions. The lack of write scalability has only occasionally been problematic, since write transactions are rarer than read transactions in many real-world workloads. Furthermore, Aurora's new multi-writer version, "Limitless", aims to alleviate this scalability bottleneck [146], albeit with poor consistency guarantees in the recommended configuration.

However, a bigger problem (which is not addressed by Aurora Limitless) has emerged with this approach: poor latency in geo-replicated scenarios. Applications are increasingly global in their design and have users across continents. By allowing only a single instance that has write access to storage, all write transactions, no matter where they initiate, must be sent to this primary instance for processing. If a client is located half-way across the world from the primary instance, that user necessarily experiences latencies in the order of hundreds of milliseconds for every write transaction. These types of latencies can be problematic for many applications [128].

In contrast, shared-nothing database systems can be extended, often in a natural and straightforward way, to perform their partitioning schemes across continents. This can potentially enable read and write transactions initiated from a location close to the partition that stores data accessed by those transactions to be processed at local (less than 10 milliseconds) latency. Both CockroachDB and YugabyteDB support

geo-partitioned transactions in this way.

For example, Fig. 3.1b shows that transactions initiating in AWS's eu-west-1 region are 3-4 times slower in Aurora than YugabyteDB when Yugabyte uses geographic partitioning to process those transactions locally, while Aurora needs to send those same transactions to its primary node in us-east-1. (See Section 3.4 for more details on the experimental setup).

These results motivate the need for a geo-partitioned version of an Aurora-style system. This chapter describes the design of such a system, called SunStorm. SunStormuses a shared-storage architecture based on the second school of thought described above, yet also partitions this storage across regions. SunStormonly needs to pay the cost of multi-partition distributed transactions for those transactions that access data at more than one region. For workloads in which such transactions are rare, the system achieves throughput of Aurora-style systems along with the latency benefits of geo-partitioned shared-nothing systems. In addition, it removes the scalability bottleneck of Aurora-style systems by allowing more than one location where write transactions can be sent.

In designing SunStorm, we followed two guiding principles: (1) minimize assumptions about the DBMS used at the compute layer, and (2) avoid cross-region communication on the critical path of transaction processing where possible.

SunStormtherefore does not assume that the DBMS assigns and stores global timestamps [24], [31], [40], [118], [144] since traditional DBMSs rarely offer such a mechanism, and implementing it may necessitate intrusive changes to their codebase.

To evaluate the contributions of SunStorm, we run experiments in which its throughput and latency are compared to leading examples built according to the two competing schools of thought that we described above. We find that in addition to the expected result (that SunStorm is able to avoid the main disadvantages of each school of thought that we described above), SunStorm achieves better latency

for geographically disperse transactions relative to systems from both schools of thought—surprisingly even the geographically partitioned shared-nothing system.

3.1 SunStorm architecture

SunStorm consists of multiple deployments of the single-primary system (described in the previous section), located in different geographic regions. The storage layers of these deployments are inter-connected to enable cross-region data sharing. SunStorm also adds a thin layer of transaction servers responsible for coordinating transactions accessing data across regions. This architecture, referred to as a **multi-primary** architecture, is illustrated in Fig. 3.2.

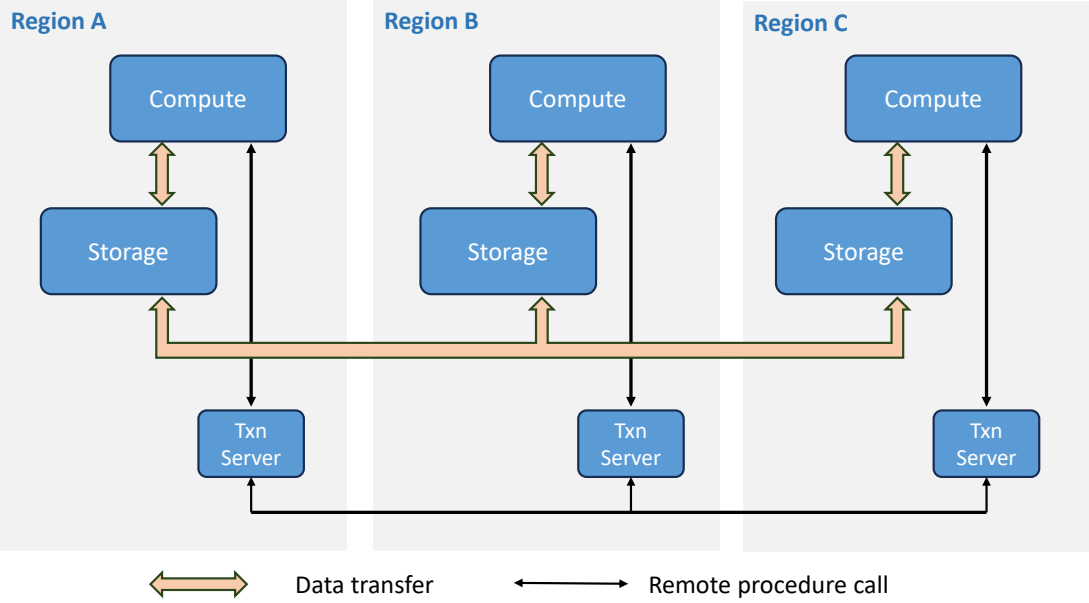


Figure 3.2: SunStorm architecture

A database consists of multiple partitions, each assigned to exactly one region as its **local data**. Partitions not local to a region constitute its **remote data**. The storage layer of each region continuously and asynchronously replicates its local data updates to all other regions. Thus, each region's storage layer stores both up-to-date local data and slightly stale remote data from other regions.

Compute nodes have complete access to read or write any data in the database as if all data was local. During transaction execution, the compute node sends data page requests to its local storage layer, which immediately returns the most recently known version of each page without communicating with other regions. While local pages are always up-to-date, the possibility of reading stale data in remote pages necessitates a multi-region commit protocol to ensure ACID-compliant transactions.

Each compute layer node runs an instance of a traditional DBMS. Any DBMS can be used if it meets the following requirements:

1. The DBMS guarantees ACID transactions.

2. Upon transaction commit, the DBMS assigns a monotonically increasing **commit sequence number** (CSN) to the transaction and stores the CSN alongside every modified data item[1].

3. A transaction can transition into a *prepared* state, after which it cannot be aborted or committed by any factor (e.g. concurrency control) other than the client.

Requirement (1) ensures ACID properties for all parts of a transaction executed within the same region.

SunStorm commits multi-region transactions using an OCC protocol. The CSNs in requirement (2) are used to establish their commit order, and relied upon during OCC validation to verify that the most recent version of a data item has been read. CSN monotonicity needs to be maintained only within each region, which can be achieved with an incrementing local counter. One such counter could be the LSN of the transaction's commit record[2].

This design decision of relying only on local CSNs instead of global CSNs has important consequences for the system's implementation. Using global CSNs is the

[1]We added this feature to the DBMS used in our initial implementation. See Section 3.3.
[2]This is the method used by our current implementation: CSNs are a subset of LSNs.

more obvious choice, and would make a globally consistent snapshot readily available to each compute node, greatly simplifying SunStorm's validation process. Nevertheless, it would incur the overhead of global coordination, potentially affecting the latency and throughput of transactions that only access data within a single region.

Requirement (3) is necessary for SunStorm's multi-region commit protocol, which is described in Section 3.2.3. It ensures that once a region votes to commit a transaction, it does not inadvertently abort the local transaction running at the compute node. Many popular database systems, such as PostgreSQL, MySQL, Oracle, and SQL Server, can satisfy this requirement with their support for X/Open XA transactions [136]. Although these systems' XA transaction implementations include recording the prepared transactions to stable storage for recovery, this is not required to meet requirement (3).

3.2 Transactions

Since SunStorm is designed not to be tied to any specific DBMS, we first define an abstract database model and then describe SunStorm's read/write operations and commit protocol based on this model.

3.2.1 Database model

We define a *database* as a set of all possible uniquely identifiable *data items*, each associated with a *value*. The value domain includes a special value $\perp$ indicating uninitialized data items. The database system supports two operations: *read*, which retrieves a data item's current value, and *write*, which updates a data item's value.

In practice, examples of data items are tuples, index pages, and tables, which can be identified by (table name, page id, offset), (index name, page id), and (table name), respectively. In our model, an insertion, update, and deletion of a data item

involves a read followed by a write operation. Specifically, an insertion reads from an uninitialized data item and writes a new value to it, and a deletion reads from an initialized data item and writes $\perp$ to it.

3.2.2 Read and Write operations

A client initiates a transaction by connecting to the primary compute node of a nearby region. Regardless of whether a page belongs to local or remote data, the compute node uses the same interface exposed by the storage layer to request the page. However, which page version to request differs between the two scenarios.

For local data, the compute node always requests the latest version. For remote data, when a transaction accesses data from region R for the first time, it requests the latest known LSN for R from the storage layer. The transaction then reuses this LSN for all subsequent requests for data from R. The compute node caches pages of both remote and local data in the DBMS's buffer pool.

It is possible for multiple versions of the same remote page to concurrently exist in the buffer pool. For instance, T1 might request a remote data page at LSN 100, and later T2 might request the same page but at LSN 200. Both versions of this page could be present in the buffer pool and read by T1 and T2 independently.

A transaction records (1) the identifiers of all remote data items that it reads along with their respective LSNs and (2) the new values of all writes to remote data items in an in-memory data structure called a **remote read/write set** (RemoteRWSet). Subsequent reads of previously written remote data items within the same executing transaction are read directly from this data structure. On commit, if the RemoteRWSet is not empty, it is sent to other regions involved in the transaction for validation and application of the new values, making the writes visible to other transactions.

Section 3.3 describes how writes to remote data are performed inside a temporary buffer locally, converted to a **logical representation**, and then sent to the owning

remote region for validation and application. By representing writes logically, index modifications need not be recorded, and remote information related to metadata stored inside the tuple header need not be coordinated across regions.

3.2.3 Commit protocol

Single-region transactions are committed using the commit protocol of the DBMS operating at the compute node using the protocol described in Section 2.2 without modification. Multi-region transactions are committed using a *decentralized 2-phase commit* (2PC) protocol [15] combined with OCC validation. A transaction is considered "multi-region" if it accesses any data from a remote region, even if all accesses are from that same region. As previously noted, SunStorm does not maintain a global snapshot, therefore, any transaction that accesses remote data, including read-only transactions needs to undergo validation for correct isolation.

Decentralized 2PC with OCC validation

This protocol is executed by the transaction servers, with communication among them being independent of communication in the storage layer. The steps of this protocol, depicted in Figure 3.3, are as follows.

1. The compute node processes the entire transaction using local and remote data, and then prepares it to commit (requirement (3) from Section 3.1) and passes its RemoteRWSet to the local transaction server.

2. The local transaction server (**initiator**) writes the RemoteRWSet to its transaction log, then sends a `PREPARE` message to all other transaction servers in the regions listed in the RemoteRWSet (**validators**). This message includes the RemoteRWSet and serves as a `YES` vote from the initiator. The initiator then waits for the votes from the validators.

3. Upon receiving a `PREPARE` message, a validator initiates a **validating transaction** on its local primary compute node.

4. The validating transaction checks for conflicts between the read data items in the RemoteRWSet and any new updates since the time the RemoteRWSet was generated, using the protocol described in Section 3.2.3. It also tentatively applies the writes from the RemoteRWSet to the database. If the validation succeeds, the compute node prepares the validating transaction and then passes the control back to its local transaction server.

5. If validation passes, the validator logs the RemoteRWSet before sending `YES` votes to all participants (initiator and validators). Otherwise, it sends `NO` votes to all participants.

6. The initiator aborts the multi-region transaction if it receives at least one `NO` vote. If it receives `YES` votes from all participants, it logs a `COMMIT` record and commits the transaction.

7. A validator aborts its validating transaction if it receives at least one `NO` vote. If it receives `YES` votes from all participants, it logs a `COMMIT` record and commits the validating transaction.

In contrast to the centralized version of 2PC [15], the decentralized protocol does not require a coordinator to collect the votes from the participants and send out commit messages. This saves one round of message passing from the coordinator to the participants [52], which significantly reduces the duration of the commit protocol in geo-distributed settings. Nevertheless, every participant sends one round of messages to all other participants, resulting in the number of messages being proportional to the square of the number of participants. This approach is taken because the number of participants (regions) is usually small.

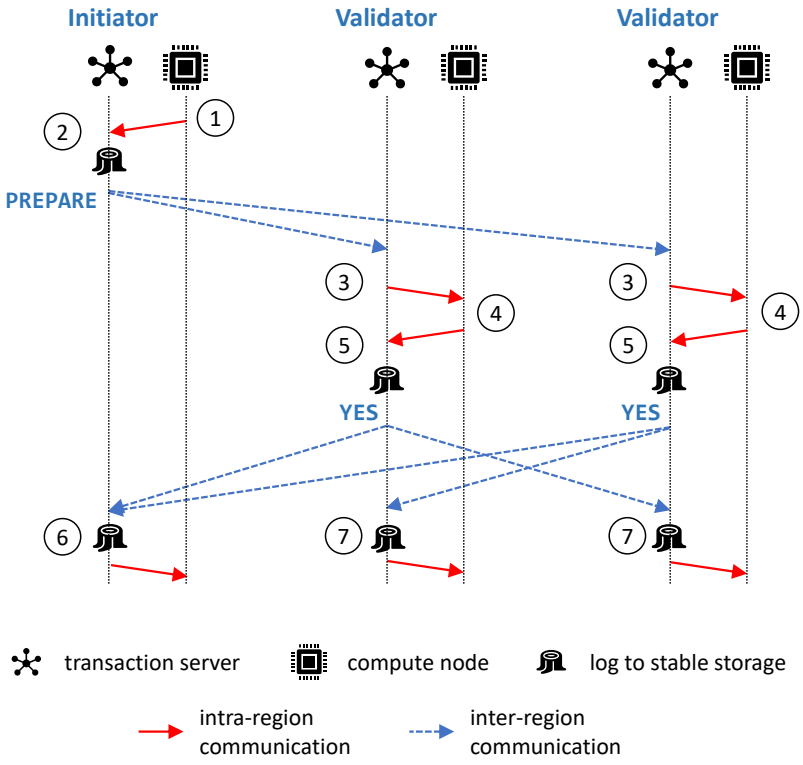


Figure 3.3: Decentralized 2PC committing a transaction

Validation

For a given remote transaction, each region only needs to validate access to its own items in the RemoteRWSet of that transaction. For each of these local data items, the validator retrieves the most recent CSN for that data item (requirement (2) from Section 3.1). This is then compared with the LSN from which the data item was read[3]. As mentioned in Section 3.1, SunStorm uses the LSN of the commit record as the transaction's CSN, allowing for a direct comparison between the LSN used by the initiator and the CSN known to the validator. The validation process fails if the latest CSN is greater than the read LSN.

Prevention of phantom reads is integrated into the validation process by including in the RemoteRWSet index pages that cover the range of a query predicate. Alternatively,

[3]Recall that even blind writes include a read in our database model

the next index key beyond the end of an index scan could be used. These approaches are parallel to the index-range locking [15] and next-key locking [84], respectively, used in lock-based protocols. We use the index-range approach in our prototype since PostgreSQL already supports index-range locking [98].

There are two potential approaches to perform validation. One option is for the transaction server to request the relevant CSN data from the storage layer and perform the validation itself. A second option is for the transaction server to construct a new DBMS transaction that performs this validation and send it to the DBMS on the compute node for processing. SunStorm uses the second approach for three reasons. First, the relevant CSN data is stored alongside other data within pages. The storage layer only knows how to return entire pages upon request. A non-trivial amount of parsing work must occur to extract the relevant CSN data from these pages. By performing this extraction effort inside the DBMS, SunStorm can take advantage of existing data deserialization, data traversal, type systems, and other code within the DBMS for this task. Second, by performing this work inside a DBMS transaction, SunStorm can leverage the DBMS's concurrency control system for transactional isolation between the validation work and other local transactions running in the validator's local compute node, avoiding the need to implement an external mechanism [66]. Third, the validation work can be combined inside the same transaction that performs the tentative local writes for that transaction so that the built-in atomicity guarantees of the DBMS can ensure that the validation and writing happen together atomically.

Recovery

If the compute node that starts a multi-region transaction fails before sending the RemoteRWSet to the initiator, or if the initiator fails before persisting the RemoteRWSet, the transaction is aborted. On the other hand, if the initiator successfully

persists the RemoteRWSet, all participants are considered equal and follow the same recovery protocol described next.

A participant may fail to receive the RemoteRWSet due to network issues or because it crashes before the `PREPARE` message arrives. Therefore, during the vote-waiting phase of the protocol, each participant periodically sends the RemoteRWSet to its peers whose votes are missing after a timeout. Since there is always at least one participant—the initiator—with the RemoteRWSet persisted at this point, every participant eventually receives the RemoteRWSet. As a corollary, if a participant fails after receiving the RemoteRWSet but before persisting it, it will receive the RemoteRWSet again after recovery and restart the validation process.

These timeouts may cause participants to receive multiple `PREPARE` messages for the same transaction. Validation transactions are idempotent, and thus can run multiple times without impacting correctness. Nonetheless, transaction IDs of recent validations can be cached in memory to avoid redundant validation effort.

The compute node and its transaction server treat each other's crashes as their own and follow a two-phase recovery protocol:

1. Upon a crash of either one of them, the compute node halts incoming traffic and aborts all ongoing transactions. If the compute node crashes, it also initiates its native recovery protocol. Traffic resumes only after the second phase described below.

2. The transaction server begins a new validation transaction for every RemoteRWSet persisted before the crash. It proceeds with the remaining steps of 2PC for these transactions as it would during normal operation. Since the transaction server persists a RemoteRWSet only after validation succeeds and there are no new transactions during the recovery phase to invalidate the RemoteRWSet, the actual validation part of these re-created validation transactions can be skipped. However,

these transactions are still necessary to re-perform the tentative writes aborted during the first phase of recovery.

3.3 Implementation

We leverage Neon [90] to build a prototype of SunStorm[4]. Neon is an open-source system based on the original single-primary version of Amazon Aurora, using a modified version of PostgreSQL at the compute layer. Our implementation builds a multi-primary, geographically replicated database system over Neon by taking advantage of Neon's data branching functionality. Each geo-partition is a separate Neon branch. Although most of our code is a layer above Neon (including the transaction server we discussed in the previous section, but excluding the recovery protocol), we did have to make changes to Neon's pageserver. For PostgreSQL, most of our changes reside within an extension. However, modifications to the PostgreSQL kernel are still necessary for SunStorm to function. In making decisions to altering it, we aimed to minimize our changes by leveraging existing facilities in the codebase. As a result, we modified approximately 3800 lines of code (with comments), among over a million lines of code, in the kernel.

3.4 Evaluation

In this section, we evaluate the performance of SunStorm while also comparing its performance to state-of-the art systems from the two existing schools of thought: the single-primary "Aurora-style" architecture and the shared-nothing architecture. For the single-primary architecture, we use Amazon Aurora PostgreSQL (referred to as Aurora for brevity) as the reference point. For the shared-nothing architecture we experimented with both CockroachDB and YugabyteDB and found their performance

[4]The source code is available at https://github.com/umd-dslam/sunstorm

to be similar (see e.g. Fig. 3.1a) which aligns with other efforts comparing the two systems [83], [106], [107]. However, CockroachDB's geo-partitioning feature is only available in their commercial enterprise version, so we focus on YugabyteDB as the reference point for the shared-nothing architecture in this section. YugabyteDB supports geo-partitioning similar to SunStorm, while Aurora does not. Furthermore, both YugabyteDB and SunStorm support multiple writer nodes, while Aurora has only a single primary node capable of handling writes. Aurora's Global Write Forwarding feature can be used to enable writes at secondary regions by forwarding write statements to the primary instance. However, only having a single writer node can be a scalability bottleneck. Although Amazon has several projects that attempt to fix the scalability bottleneck—Aurora Multi-Master and Aurora Limitless—the former has been discontinued [8] and the latter is not generally available at the time of this writing, and thus not available for our experiments[5].

3.4.1 Experimental setup

We ran our experiments using 3 AWS regions: North Virginia (us-east-1), Ireland (eu-west-1), and Tokyo (ap-northeast-1) whose round-trip time is shown in Table 3.1.

Table 3.1: Round-trip time of all region pairs (ms)

	ap-northeast-1	eu-west-1
us-east-1	142	68
eu-west-1	198	

In each region, we deployed YugabyteDB in a three-node cluster of c5d.4xlarge EC2 instances[6], which is among the recommended instance types for this system [145]. SunStorm, with separate compute and storage layers, is architected differently from YugabyteDB. To generate a reasonable comparison despite the different architecture,

[5]As noted earlier, Limitless only addresses the scalability bottleneck of the single writer node. It does not solve the latency bottleneck of forwarding writes to a primary region.

[6]These instances run on Intel Xeon Platinum 8000 series processors with clock speeds of 3.6GHz, and have 16 vCPUs with 32GB of memory, and a 400GB NVMe-based SSD.

we used YugabyteDB's hardware as a baseline for cost and selected appropriate instance types for the different components of SunStorm without exceeding that cost. Similar to SunStorm, Aurora also has separate storage and compute layers. Information regarding the hardware used for Aurora's storage layer is not publicly accessible. For the compute layer, Aurora allows users to select the instance type of the compute node from a limited list of options. We opted for the db.r5.4xlarge instance type, which most closely matches the compute node specifications of the other systems. We summarize our hardware choices in Table 3.2.

Table 3.2: Instance types per system in each region. Numbers in parentheses are the instance quantities.

	SunStorm	**YugabyteDB**	**Aurora**
Compute	c5.4xlarge (1)	c5d.4xlarge (3)	db.r5.4xlarge (1)
Page store	c5d.4xlarge (1)	N/A	Unknown
WAL store	c5.2xlarge (3)		

We used BenchBase (formerly known as OLTPBench) [38] to generate the workloads. We deployed one client machine in each region and provisioned enough capacity for them to avoid being a bottleneck. Every client thread issues transactions in a closed-loop. A transaction can either be a single-region (SR) or a multi-region (MR). Both SR and MR transactions always access at least one data item in the local region, while MR transactions access at least two regions. Aurora does not partition data across regions and simply forwards write statements to the primary region, which is us-east-1 in all experiments. We executed the transactions at the highest possible isolation level for all systems; however, Aurora only allows running at the `REPEATABLE READ` isolation level in the secondary regions with write forwarding. When serving reads from replicas, Aurora supports either strong consistency (`GLOBAL`) and eventual consistency (`EVENTUAL`). We experiment with both options.

We evaluated the systems using a microbenchmark based on the Yahoo! Cloud

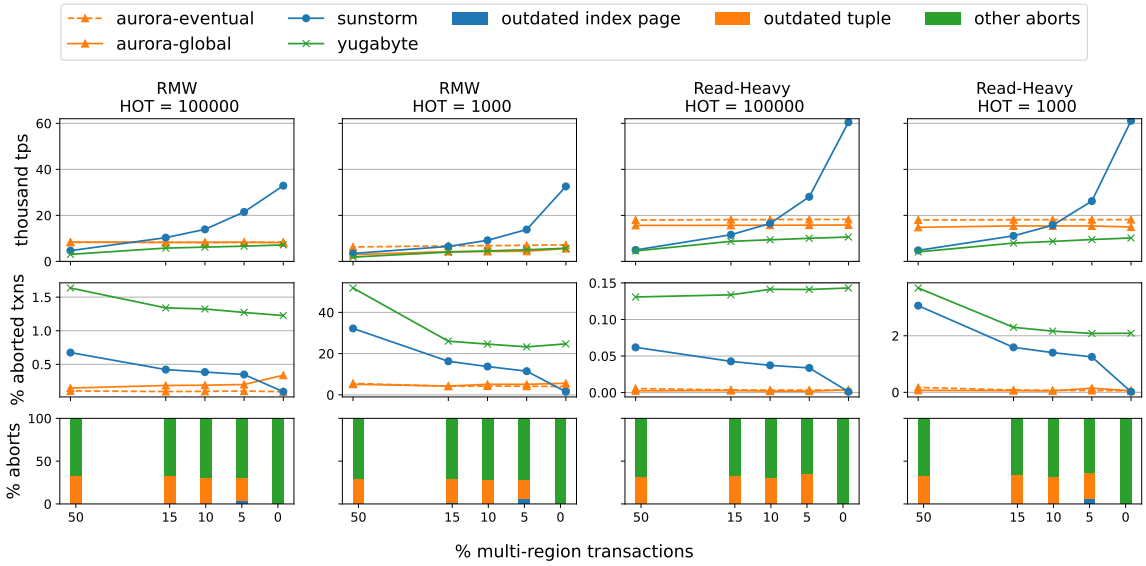


Figure 3.4: Microbenchmark performance

Serving Benchmark (YCSB) [30], adapted for transactions. The benchmark involves one table with 11 columns. The first column, containing 64-bit integers, serves as the primary key, while each of the other columns consists of 10 random bytes. The table is partitioned over the regions, with each partition containing one million rows.

Each transaction accesses 8 rows from the table without retries in case of aborts due to error. We used three YCSB workloads: *Read-Heavy* (95% read, 5% write), *RMW* (50% read-modify-write, 50% read), and *Read-Only* (100% read). For an MR transaction, the number of regions it accesses follows a Zipfian distribution and the accessed rows are divided evenly among the regions. The data within each region is separated into a *hot set* and a *cold set*. Every transaction accesses one row from the hot set in each involved region. We define `HOT` to be the number of tuples in the hot set at each region. Consequently, contention increases as the `HOT` value decreases.

For the first set of experiments, we varied the % MR parameter at 0%, 5%, 10%, 15%, and 50%, each of which under two `HOT` settings: 100,000 and 1,000. Fig. 3.4 summarizes the results of the experiments.

3.4.2 Throughput

The throughput results are presented in first row of Fig. 3.4. To better understand the effect of the consistency parameter, we executed the Aurora experiments with both `GLOBAL` and `EVENTUAL` consistency settings.

Aurora's throughput stays flat since it does not support geo-partitioning and is thus unaffected by the %MR parameter. However, its throughput is limited by having only a single writer-node and is therefore bottlenecked by the speed at which this node can perform writes. Thus its throughput is much higher for the read-heavy workload than the *RMW* workload. The throughput for the two consistency settings is very similar since the primary cost of improved consistency is latency (not throughput). However, the `GLOBAL` workload performs additional work to verify that the secondary is up-to date with all the changes committed at the primary as of the moment when a transaction began. Therefore, the secondary nodes notice a slight drop in throughput with the `GLOBAL` setting. This drop is noticeable in the *Read-Heavy* workload because the secondary nodes that service the *read-only* transactions carry out additional checks before returning results to the clients.

YugabyteDB supports geo-partitioning and therefore its throughput increases with better partitioning and fewer MR transactions. However, its overall throughput is limited by its shared-nothing architecture. Since it partitions data evenly across all nodes within a region, it must pay distributed processing, coordination, and commit costs even for SR transactions, as multiple nodes within that region may store data accessed by that transaction.

In contrast, SunStorm only pays distributed processing, coordination, and commit costs for MR transactions. It therefore achieves a large jump in throughput where there are fewer MR transactions since it gets the benefit of having multiple writer nodes (one per region) without paying the cost of coordination across regions.

Thus for workloads that partition well across regions, SunStorm yields all the

benefits of geographic partitioning while also yielding the benefits of the Aurora-style approach of avoiding distributed processing of transactions. However, for workloads that do not partition well, standard Aurora is preferable (especially for read-heavy workloads in which the write-scalability bottleneck is less problematic), since it requires no cross-region coordination, while SunStorm requires cross-region communication, validation, and two-phase commit. Furthermore, as contention increases, abort rate increases, which also reduces throughput. We analyze the abort rate more deeply in the next subsection.

Irrespective of the workload, at MR = 0%, SunStorm yields more than three times the throughput of Aurora, despite only having three times the processing resources for write transactions because all writes in Aurora are processed at a single node. Although the SR transactions within a region do not conflict with SR transactions from another region, they still contend with each other for shared database resources, such as the global data structure that stores the set of active transactions and the writer thread that flushes transaction log records. Therefore Aurora's design of processing all writes at a single primary creates resource contention while serving a larger traffic, leading to a throughput drop.

3.4.3 Abort rate

The second row in Fig. 3.4 illustrates the abort rate as a percentage of all submitted transactions. DMBS-initiated aborts in SunStorm may occur due to the DBMS's native concurrency control or failed validation (in case of MR transactions). In contrast, Aurora only aborts transactions due to the DBMS's concurrency control mechanism.

SunStorm and the eventual consistency version of Aurora abort SR transactions (MR = 0%) at similar rates. However, the `GLOBAL` consistency version of Aurora faces a higher abort rate on the *RMW* workload due to increased latency, resulting in an

extended conflict window. YugabyteDB experiences a higher abort rate for all cases of SR transactions. This is again due to YugabyteDB's shared-nothing architecture, which causes even SR transactions to require distributed coordination across the multiple nodes within that region. This coordination increases latency, extends the conflict window [123], and thus causes aborts in YugabyteDB's OCC protocol.

Similarly, MR transactions in SunStorm and YugabyteDB are more susceptible to aborts because they take longer to complete. While their overall abort rates are lower than 2% at low contention, they are much higher at high contention since the probability of a conflicting transaction existing during the OCC contention window is high. SunStorm's abort rate is generally lower than YugabyteDB's because it requires fewer rounds of communication in MR transactions. As discussed in Section 3.2.3, SunStorm only communicates with other regions at commit time, and uses a decentralized 2PC protocol that requires a single communication round-trip between the initiator and the validators. A shorter transaction and commit duration leads to a smaller window where an uncommitted remote transaction can cause other transactions to abort. In Aurora, transactions in the MR category are not real MR transactions since all transactions are processed by the primary region. Aurora's abort rate increases with % MR because MR transactions access more than one hot key, thus are more likely to conflict with other transactions.

For the Read-Heavy workload, all three systems have a low abort rate ($\leq 4\%$), irrespective of the contention level. This is because only a small percentage of transactions update the data, which is ideal for OCC protocols [68].

3.4.4 Abort reasons

To better understand the effects of SunStorm's validation mechanism, we measured the different causes of aborts for SunStorm transactions. We track three main causes: (1) outdated index pages, (2) outdated tuples, and (3) other aborts by PostgreSQL.

The validation process causes (1) and (2), while aborts due to (3) could occur on a single-node PostgreSQL deployment. We did not encounter any aborts by distributed deadlocks in our experiments. The results are shown in the last row of Fig. 3.4. As expected, validation aborts take a greater share of abort percentage as the %MR increases.

Under low contention, both *RMW* and *Read-Heavy* workloads only experience outdated tuple aborts and PostgreSQL-initiated aborts. SunStorm does not encounter any outdated index aborts due to the *Heap-Only-Tuple* optimization employed by PostgreSQL. This optimization enables PostgreSQL to skip index updates whenever a tuple update does not involve indexed columns and the old tuple's page contains sufficient space for the new tuple. Given that updates are spread out across the dataset under low contention, the updated tuples are likely to be placed on the old page, avoiding index updates. Under high contention, both workloads also do not or rarely see aborts due to outdated index page, but for different reasons. Most aborts happen because of the SSI protocol in PostgreSQL or because of conflicts between SR transactions and prepared MR transactions.

3.4.5 Read-Only throughput

Next, we examine the systems on a Read-Only workload. The throughput results are shown in Fig. 3.5. Aurora with `EVENTUAL` consistency achieves the highest throughput because the secondary regions operate almost independently in the absence of writes and its poor consistency guarantees gives it a large advantage relative to the other systems. However, when it uses `GLOBAL` consistency (which is more comparable to the guarantees of the other systems), it runs much slower because the secondary node needs to contact the primary at the start of every transaction and waits for it to respond with the last persisted LSN, to make sure that the reads are consistent. SunStorm's throughput at MR= 50% is comparable to the Read-Heavy workload,

as all MR transactions are validated even if they are read-only (see Section 3.2.3). However, at MR= 0%, SunStorm's throughput matches Aurora under `EVENTUAL` consistency because then it too can run completely without any coordination across or within regions. YugabyteDB's performance is similar to its performance on the Read-Heavy workloads since Read-Only transactions still follow the same execution protocol and require coordination within a region.

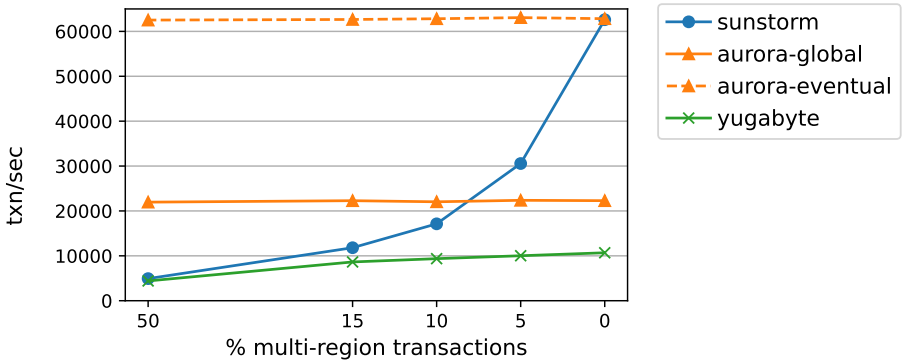


Figure 3.5: Read-Only workload throughput (HOT = 1000)

3.4.6 Latency

In Fig. 3.6, we plot the latency data of the systems across the three regions for `HOT` = 100000 (top two rows) and `HOT` = 1000 (bottom two rows), representing low and high contention cases, respectively, for the $MR = 5\%$ and $MR = 50\%$ workloads. The x-axis categorizes transactions as n-regions and the y-axis plot the latency in milliseconds. "1-region" refers to SR transactions, while "2-regions" and "3-regions" refer to MR transactions that access data in 2 regions and 3 regions, respectively.

To summarize the latency, we use a modified box plot, where the upper whisker represents the 99^{th} percentile latency instead of the usual 1.5 times the interquartile range (the lower whisker is the normal 1.5 times interquartile range).

Aurora exhibits very low latency in its primary region (us-east-1) as expected, and much larger latency in the secondary regions due to its protocol of forwarding writes

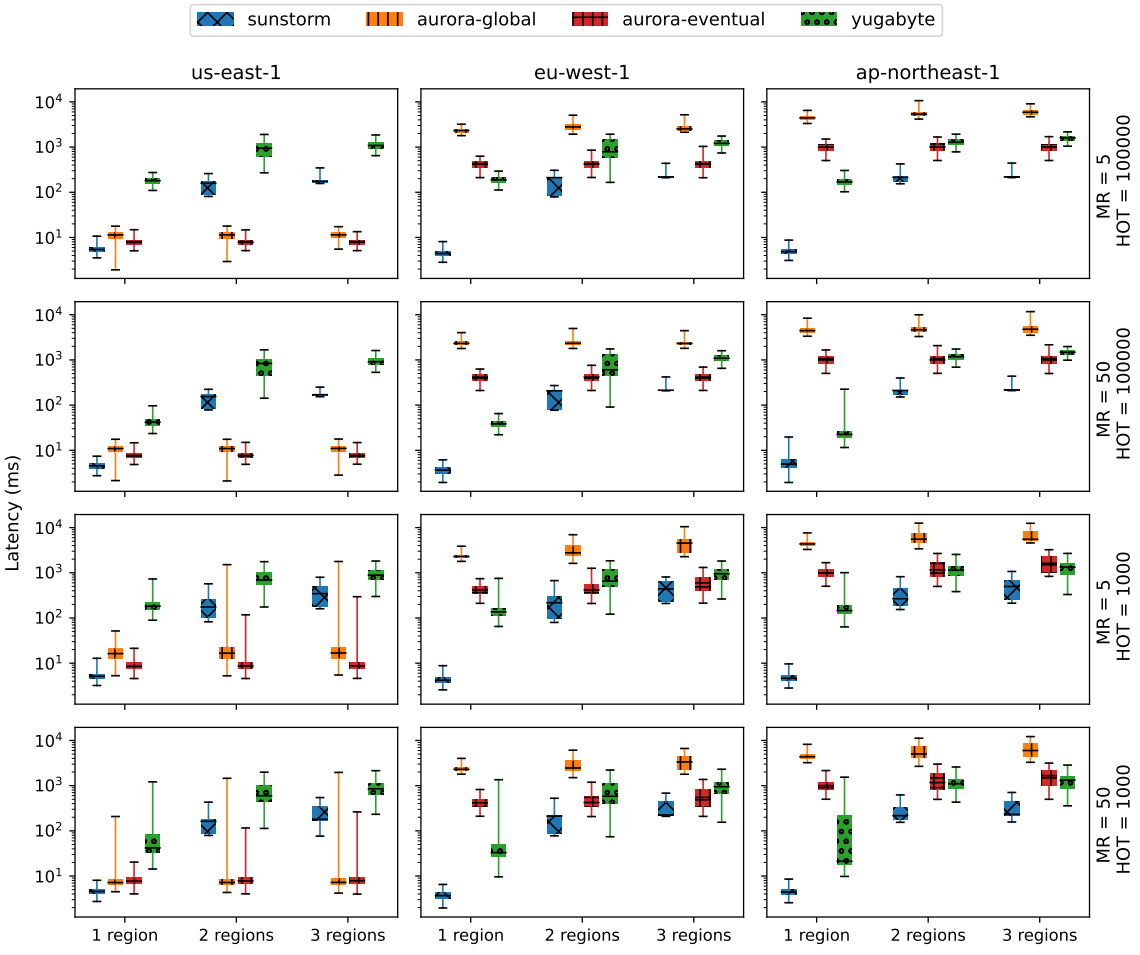


Figure 3.6: Microbenchmark Latency (RMW)

to the primary region. Additionally, when Aurora uses the `GLOBAL` consistency setting, its secondary nodes incur a cross-region round-trip to obtain the latest state of the primary at the start of each transaction. Thus, transactions initiated there experience higher latency under `GLOBAL` consistency compared to the `EVENTUAL` setting.

In contrast, SunStorm and YugabyteDB leverage data access locality through geo-partitioning, achieving lower latency for SR transactions (the "1 region" columns) across all regions. In Aurora's primary region, SunStorm's SR transactions have slightly lower latency than Aurora's, since they do not have to compete for processing resources with transactions from other regions, while in Aurora all transactions are processed by the same primary region. YugabyteDB also employs geo-partitioning, but has a much higher latency baseline due to its shared-nothing architecture. In YugabyteDB, even SR transactions are processed as distributed transactions across the nodes in that region and must pay the latency associated with distributed execution and commit.

The latency of MR transactions in SunStorm is close to a single round-trip to the furthest participating region. This is because SunStorm accesses local data during transaction execution and communicates across regions only for commit validation, which involves only one round of communication. Conversely, YugabyteDB needs to communicate with other regions for reads of remote data, resulting in possibly multiple round-trips within a transaction.

High contention adversely affects SunStorm's latency, especially at the tails. This is because some SR transactions may conflict with and end up waiting behind slower MR transactions, raising their latency to match that of those MR transactions. While this phenomenon may also occur in YugabyteDB, the impact is less noticeable since all transactions in YugabyteDB generally have much higher latency. Similar to the results from the previous subsection, Aurora's numbers are again only affected by running over multiple regions to the extent that MR transactions may access more

hot records. In the primary (`us-east-1`) region, Aurora with the `GLOBAL` setting experiences higher tail latency because the local transactions are more likely to be blocked by the slower transactions originating in the remote regions.

3.5 Conclusion

Experiments show that SunStorm performs best when the quality of geographic partitioning is high and most transactions only access data from a single region. In such a scenario, SunStorm achieves all the advantages of Aurora-style systems that do not require distributed transaction processing or coordination within a region, while also achieving much improved latency relative to the standard single-primary architecture since transactions can run locally to the client without having to ship all transactions to a primary region.

Chapter 4

Detock: High performance multi-region transactions at scale

Modern data stores typically replicate data for improved availability, durability, read throughput, and/or latency. Data stores designed for global applications typically replicate data across large geographic distances, which further improves robustness to region failure, and can allow reads to occur locally to an application client.

Data stores that allow replicas to temporarily diverge in a manner visible to the client are termed *w*eakly consistent, and those for which such divergence does not exist or is kept invisible to the client are termed *s*trongly consistent. The gold standard for strongly consistent guarantees in the context of transactional systems is strict serializability [14], [17], [55], [94], which ensures that transactions submitted after earlier transactions complete never observe a state prior to those completed transactions, even if they are processed on a different replica.

There are two common approaches for implementing strict serializability in practice. The simplest approach is to have a primary copy of each data item. All writes are performed by that primary copy, and strongly consistent reads are directed either to that primary copy or other copies that are replicated to synchronously from that

primary copy [33], [74], [100], [101]. The second approach allows writes to occur at any replica, but performs a consensus protocol to avoid replica divergence [13], [31], [118], [123], [138], [149].

Both of these approaches can support geographic partitioning of data for improved performance. In the first approach, the primary copy of different data items can be stored in different regions [29], [33], [74]. In the second approach, separate consensus groups can be formed in different regions [109], [118]. Either way, geo-partitioning decreases latency of transactions that initiate near the region of their accessed data and can therefore increase the overall performance of a workload if such transactions are prevalent.

Unfortunately, even for workloads where such transactions are common, there still exist some transactions which must access data in more than one partition. Such transactions necessarily require coordination across partitions (and therefore across geographic regions) to ensure strict serializability, and this increases latency. The bigger problem, however, is that when they conflict with single-partition transactions, they become hard to complete: optimistic concurrency control approaches result in extremely high abort rates under high contention, and pessimistic approaches result in high amounts of deadlock. Even single-region transactions can end up getting involved in deadlocks or OCC aborting because of the presence of these slow multi-region transactions. Therefore, it is extraordinarily difficult to achieve high throughput under high contention and a non-trivial number of multi-region transactions.

One approach to avoiding these issues is to use deadlock avoidance techniques to enable pessimistic concurrency, providing high-throughput transaction processing under high contention. For example, the work on SLOG creates separate consensus groups per region, geographically partitions data across these regions, and runs every multi-region transaction through a global ordering mechanism to completely avoid deadlock [109]. However, this approach adds the latency of the global ordering layer

in addition to the latency required for coordination during normal processing of strictly serializable multi-region transactions, further impacting the performance of conflicting single-region transactions.

Instead, in this chapter, we present a new graph-based concurrency control protocol that enables multi-region transactions (in addition to single-region transactions) to be scheduled deterministically at each region such that all regions involved in processing a transaction will construct the same graph independently and process transactions completely without cross-region coordination after receiving all parts of the transaction. The graphs constructed by each region are formed based on conflicting accesses by different transactions, and indeed may contain deadlocks. However, since each region constructs the same graph, deadlocks can be resolved by dynamically reordering accesses by deadlocked transactions to resolve the deadlock **deterministically** without ever having to resort to aborting transactions and without having to communicate this reordering with other regions.

Nevertheless, high network delays between regions can cause the size and number of deadlocks to grow unbounded. We therefore implement a practical version of this algorithm within a new system called DETOCK that annotates transactions with real-time based timestamps, which are used to strategically schedule transactions to reduce the probability of deadlock. DETOCK also implements a novel protocol for migrating data to other geo-partitions using a simpler approach than used in previous work [109].

When comparing DETOCK to several alternative state-of-the-art systems that support geographic partitioning such as SLOG and CockroachDB, we find that DETOCK can lessen throughput reductions caused by multi-region transactions under high conflict by several orders of magnitude, while simultaneously reducing latency by avoiding unnecessarily cross-region coordination.

4.1 System architecture

DETOCK is a geo-partitioned database system, and uses a similar high-level architecture to recent state-of-the-art geo-partitioned database systems, while introducing novel approaches to concurrency control, deadlock resolution, and data migration. This section overviews the basic architectural approach that DETOCK shares with other geo-partitioned database systems — most notably SLOG and CockroachDB — and we defer the discussion of the unique aspects of DETOCK's design to the following section.

The system is deployed across multiple geographic regions. A *region* consists of servers connected via a low-latency network. These servers typically reside within a single data-center or multiple data-centers that are in close proximity with each other. Each item is assigned to exactly one geographic region. This is called a *home region* in both SLOG and CockroachDB. The identifier of the home region for a data item is stored in the header of that data item.

A region may store *local data* for which it is designated as the home region, and *remote data* which are materialized by replaying logs asynchronously replicated from other regions. Data is also partitioned locally within a region independently of home status: each partition might contain a mix of local and remote data.

Similar to SLOG, DETOCK relies on deterministic transaction execution to substantially reduce cross-region coordination. Fig. 4.1 shows the architecture of this style of deterministic system in a deployment over two regions A and B. Clients send transactions to their closest region. The first server that receives a transaction becomes its *coordinator*, which first resolves non-deterministic commands in the transaction (e.g. `random()` and `time()`), then attempts to extract its read/write set. When this is not possible via static analysis [59], the OLLP protocol is used, which obtains an initial estimate of the transaction's read/write set via a reconnaissance query [123]. Each region maintains a distributed index called a *Home Directory* that contains the

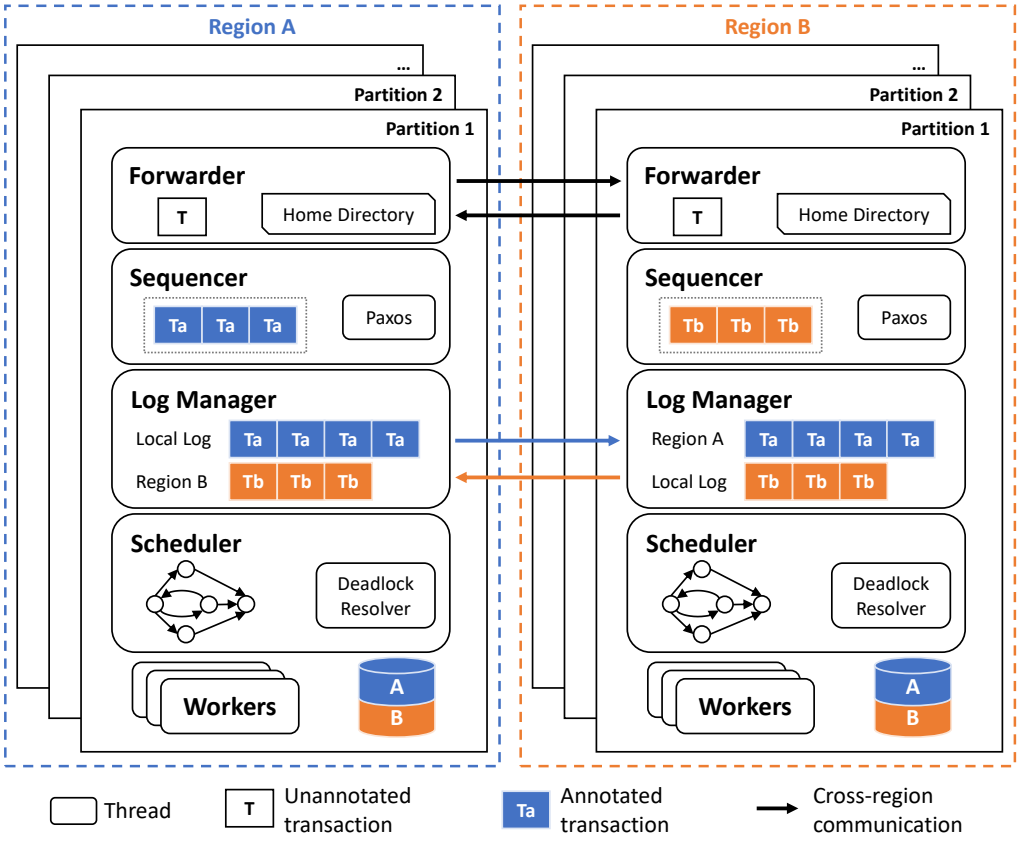


Figure 4.1: Architecture of Detock

cached value of the current home for each known data item. The Forwarder of the coordinator uses this index to augment the read/write set with the home information of every data item. It then annotates the transaction with this augmented read/write set and forwards the transaction to its home region(s).

We denote in Fig. 4.1 annotated transactions housed by region A as Ta, and by region B as Tb. Once these transactions reach their home regions, they are put into batches and inserted into a Paxos-maintained *local log* by the Sequencers. This log is synchronously replicated within a region to tolerate failure of individual servers, and optionally to nearby regions to increase availability during (rare) failures of an entire region.

A region can deterministically replay a local log from any other region R to obtain the state of R's local data. Therefore, persisting the local logs is sufficient for durability, and replication is performed by shipping these local logs. While it is not required for a region to hold any remote data, having a possibly stale copy of the remote data allows local snapshot reads of data from other regions and makes executing multi-home transactions (including the home-movement transactions in Section 4.3) faster. To this end, regions in DETOCK and SLOG **asynchronously** exchange their local logs to each other, so each region eventually receives and replays the complete local log from every other region, as can be seen in the Log Managers of both regions in Fig. 4.1.

To replay the logs, a Scheduler constructs a dependency graph for the transactions in the logs with the help of the Deadlock Resolver (Section 4.2.2), and schedules them to be executed by the Workers.

If all data accessed by a transaction belong to a single region, it is called a *single-home* transaction; otherwise, it is *multi-home*. Multi-home transactions insert records into the local logs of each home region for the data accessed by that transaction. As mentioned above, SLOG globally orders multi-home transactions to avoid inconsistently ordering them across regions (e.g. T1 before T2 at region 1, but T2 before T1 at

region 2) which could result in serializability violations, OCC aborts, or deadlock [109]. DETOCK eliminates this global ordering, but must therefore deal with the problems that arise from inconsistent ordering (discussed in the next section). By eliminating multi-home transaction ordering, DETOCK is able to guarantee that each transaction, regardless of its type, only needs **a single-round trip** from the initiating region to the participating regions: the initiating region sends the multi-home transaction to each region which houses data that it accesses, waits to receive the local log records back from those regions through which it can derive the state at those regions over which that transaction must run, and can then process that transaction to completion locally. Every other region, including the ones that write local data, see the same local logs and also process that transaction locally.

4.2 Transaction processing

When a new transaction arrives at the system, its coordinator invokes the function `StartTxn` shown in Algorithm 1. Although most deterministic systems such as SLOG and Calvin do not require assigning a globally unique identifier to a transaction upon arrival, most other highly consistent ACID-compliant distributed systems — including CockroachDB and Spanner — give transactions globally unique identifiers. DETOCK takes the latter approach despite being a deterministic system since the identifier will be used in the concurrency control protocol. The globally unique ID is generated by concatenating (in binary) a local transaction counter with a globally unique ID statically assigned to the coordinator's server (Line 2). The Home Directory is used to find the cached values of the home regions for all data items in the read/write set (Line 3-11).

The number of unique home regions retrieved is used to determine whether the transaction is single-home or not (Line 13-14). Incorrect read/write sets (from the

Algorithm 1: Starting a new transaction

```
1  function StartTxn(txn)
2      txn.id = new globally unique ID
3      if txn.isHomeMovement then                                  /* see Section 4.3 */
4          key = txn.movedKey
5          txn.oldHome = HomeDirectory(key)
6          txn.homeInfo = {(key, txn.oldHome), (key, txn.newHome)}
7      else
8          txn.homeInfo = ∅                                  /* set of (key, region) pairs */
9          foreach key in txn.readSet ⋃ txn.writeSet do
10             Add HomeDirectory(key) to txn.homeinfo
11         end
12     end
13     regions = unique regions in txn.homeInfo
14     txn.isMultiHome = size of regions is larger than 1
15     if txn.isMultiHome then
16         /* oneway[r] is estimated one-way network delay to region r              */
17         maxOneWay = Max(oneway[r] : ∀r in regions)
18         txn.timestamp = Now() + maxOneWay + overshoot
19     end
20     Call AppendLocalLog(txn) for every region in regions
```

OLLP protocol) or home information (from stale values in the Home Directory) are deterministically detected later during execution and cause the transaction to abort and restart. However, these restarts are expected to be uncommon in practice: OLLP aborts only occur when the access set of data depends on the current state of the database [111], [123], while home information aborts only occur for a short period of time after data is rehoused in a different region. The transaction is then forwarded to the participating regions (Line 20). [The timestamp assigned in Line 18 is an optimization that is described in Section 4.2.2.]

4.2.1 Single-home transactions

The initial steps of transaction processing of single-home transactions DETOCK are identical to those of SLOG: When a single-home transaction reaches a node at its presumed home region, the Sequencer of that node runs the code in Algorithm 2,

Algorithm 2: Appending transactions to the logs

```
1  function AppendLocalLog(txn)
2      localTxn = txn
3      if txn.isMultiHome then
4          Sleep until Now() ≥ txn.timestamp
5          exclude = keys in txn.homeInfo where region ≠ curRegion
6          localTxn.readSet = localTxn.readSet \ exclude
7          localTxn.writeSet = localTxn.writeSet \ exclude
8      end
9      Append localTxn to batch
10 upon batch is ready do
11     pos = Append batch to local Paxos log and get its position
12     Asynchronously call AppendGlobalLog(curRegion, pos, batch) for every region
13     batch = ∅
14 function AppendGlobalLog(reg, pos, batch)
15     localLogs[reg, pos] = batch
16     lastPos = position of the last batch in localLogs[reg] that was added to globalLog
17     while localLogs[reg, lastPos + 1] ≠ null do
18         foreach txn in localLogs[reg, lastPos + 1] do
19             Append txn to globalLog
20         end
21         lastPos = lastPos + 1
22     end
```

which puts the transaction into an in-memory batch along with other concurrent transactions that arrive at the same node (Line 9). The size of the batch window is configurable, and defaults to 5ms. The Sequencer then appends the batch to the region's local input log via Paxos (Line 11). After this point, the transaction is durably logged for recovery. The position in the Paxos log, along with the contents of the batch is asynchronously replicated from that region to every other region, such that every region eventually receives the complete set of ordered batches from every other region (Line 12). DETOCK and SLOG also support synchronous replication to near-by regions for improved robustness to region-failure.

At each region, the Paxos logs from all regions (including its own) are interleaved arbitrarily by the Log Managers to form that region's view of the *global log* (Line 15-22). Each region may interleave transactions from different local logs into the global log in different ways; however, if transaction B is after A in any region's local log, B will be after A in every region's global log, since logs records from the same region are numbered by that region and never reordered.

From this point forward, DETOCK's processing of single-home transactions differs from SLOG. In both systems, each region executes all transactions found in its global log in parallel, but in a manner equivalent to if they had been executed sequentially in log order. However, SLOG uses a locking based mechanism to achieve this, while DETOCK uses an approach based on dependency graphs in order to facilitate deadlock detection and resolution. Algorithm 3 presents the pseudocode of the Scheduler where the dependency graph is constructed (Line 2-7).

Definition 4.2.1 *Two transactions T_i and T_j are said to conflict on a tuple (d, r), where d is a data item and r is a region, if both transactions access d, at least one of the transactions writes to d, and both of them expect r to be d's home region.*

A dependency graph is a directed graph where vertices correspond to transactions, and an edge (T_i, T_j) exists if and only if:

Algorithm 3: Scheduling transactions for execution

```
1  upon a new transaction txn is appended to globalLog do
2      newEdges = ∅
3      foreach (k, r) in txn.homeInfo do
4          prev = latest transaction that conflicts with txn on (k, r)
5          if prev ≠ null then  Add (prev.id, txn.id) to newEdges
6      end
7      Add txn.id and newEdges to dependency graph 𝒢
8      Broadcast txn.id and newEdges to all local partitions
9  upon a transaction txn becomes ready; in a Worker thread do
10     if not txn.isHomeMovement then
11         foreach (key, region) in txn.homeInfo do
12             if region ≠ storage.getHome(key) then  Abort txn
13         end
14     end
15     else if txn.oldHome ≠ storage.getHome(txn.movedKey) then
16         Abort txn
17     end
18     Execute the code in txn
19     Remove txn.id and its associated edges from 𝒢
20 /* Called periodically in a background thread                    */
21 function FindAndResolveDeadlocks()
22     𝒢′ = FindStableSubgraph(𝒢)          /* see Section 4.2.2 */
23     foreach scc in FindSCCs(𝒢′) do
24         Deterministically serialize scc in 𝒢
25     end
```

- T_i is at a position earlier than T_j in the global log[1], and
- There exists a tuple (d, r) such that both T_i and T_j conflict on (d, r), and there does NOT exist a transaction T_k such that T_k is between T_i and T_j in the global log and T_k conflicts with both T_i and T_j on (d, r).

Despite each region having slightly different versions of the global log, they are all guaranteed to (eventually) construct the same dependency graph, since the definition of conflict prevents conflicts across regions, and thus the order of interleaving logs from different regions does not impact the ultimate structure of the dependency graph. Therefore, each region can process the transactions in its global log independently,

[1]Traditionally, a wait-for graph is constructed with directed edges pointing away from the waiting transaction. However, reversing the edges simplifies our implementation.

without any communication with other regions, and arrive at the same final state.

If all transactions are single-home, the dependency graph constructed from the global log will be a directed acyclic graph (DAG). This is because edges can only arise among transactions within a region, which are strictly ordered. Therefore, processing of the transactions follows the topology order of that graph. A finished transaction is removed from the graph along with its outgoing edges. A transaction is executed only when there are no more incoming edges pointing to it. Different partitions in the same replica may only see partial views of the DAG. However, since there is no cycle in a DAG (because all transactions are single-home), the transactions can be processed without a distributed deadlock detection mechanism.

Transactions accessing multiple partitions in the same replica follow a deterministic execution protocol similar to Calvin [123] and thus do not require two-phase commit. Unlike Calvin, before accessing a data item, a transaction needs to check whether the home region identifier stored alongside the data item matches with the expected home region retrieved previously from the Home Directory. If they do not match, the transaction must be aborted and restarted (Line 10-17). Since all regions eventually receive the same set of logs and the transactions are processed deterministically, the regions apply the same sequence of updates to each data item. The home region identifier, being part of a data item, is thus updated at the same point in that update sequence. Consequently, all regions make the same decision as to whether to abort a transaction or not based on the comparison between the actual home region identifier stored alongside the data item and the assumed home region identifier stored in the transaction. Once a transaction finishes its execution, the scheduler removes it from the graph and schedules transactions that become ready as a result of this removal (Line 18-19).

Fig. 4.2 shows an example of single-home transaction processing. There are 3 regions: US, EU, and AP, each of which holds a complete copy of the database. In

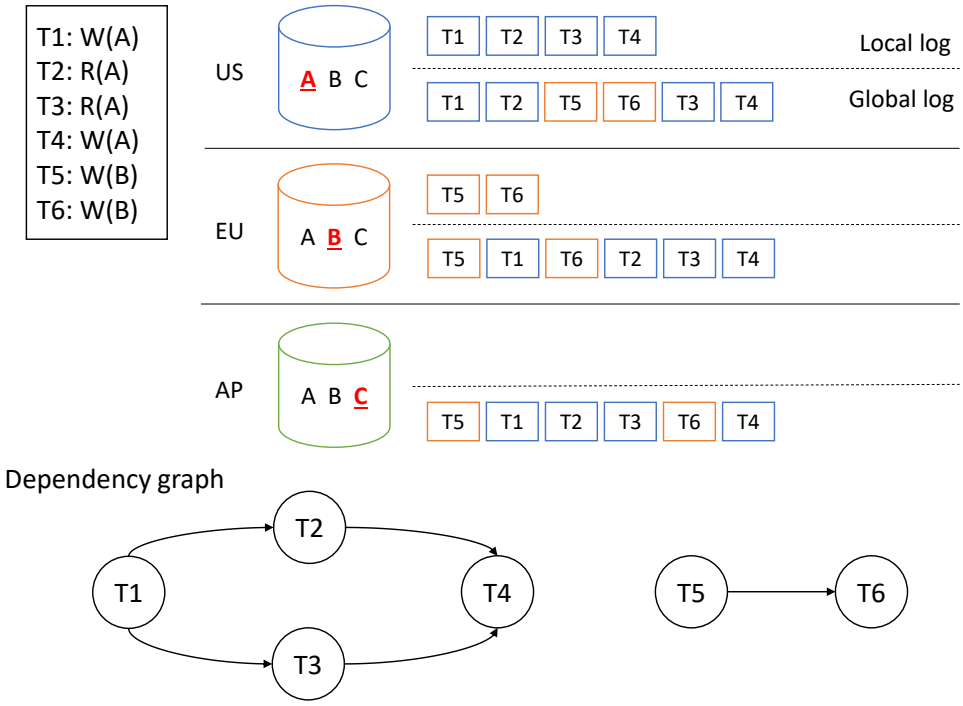


Figure 4.2: Single-home transaction processing

this example, there is only one partition and one replica in each region. The local data of each region is shown in red and underlined. The 4 transactions T1-T4 access data item A so they are ordered in the local log of the US region. Transactions T5 and T6 access data item B so they are ordered in the EU region. Each region eventually obtains the local log from every other region and interleaves them to form its view of the global log. As noted above, the generated dependency graphs for the regions eventually become identical, despite each region having a different version of the global log. The associated home regions of the data items are not changed in this example, so conflict of two transactions is determined based solely on their read and write operations on the data items, shown on the top left of the figure.

4.2.2 Multi-home transactions

When a multi-home transaction reaches a participating region, it follows a different protocol than that of a single-home transaction. Each region uses the home information

stored in the transaction to generate a special kind of transaction called a GraphPlacementTxn that contains a list of the keys from the original multi-home transaction that are local to the current region (Algorithm 2, Line 3-8). One GraphPlacementTxn per transaction, designated by the coordinator, also contains the original code for that transaction.

GraphPlacementTxns are initially treated like single-home transactions: they are put into the local logs at their home regions, make their way to the global logs through local log replication, and finally get added to the dependency graphs at every region.

For each transaction, T, each region will eventually receive GraphPlacementTxns from each region that the coordinator expected to house relevant data. The first GraphPlacementTxn for T that is placed in a region's global log causes a new vertex representing T to be created in that region's graph. Subsequent GraphPlacementTxns of T share this same vertex. Edges created by these GraphPlacementTxns are added to that vertex.

GraphPlacementTxns establish an order between multi-home and single-home transactions at the region that generated the GraphPlacementTxn. However, they do not globally order multi-home transactions, since two different regions may generate GraphPlacementTxns for a set of multi-home transactions in different orders. There is thus a concern that the generated graph may contain cycles, which would lead to deadlock during processing.

Fig. 4.3 shows an example scenario that would lead to deadlock, using the same setup as Fig. 4.2. Three multi-home transactions arrive at the system: T1 and T2 both access data item A and B, so they are sent to the US and EU regions. T3 accesses data item A and C, so it is sent to the US and AP regions. All accesses are read-modify-write operations. Each region generates the GraphPlacementTxns for the multi-home transactions and places them in its local log; however, the order they are placed differs across regions. Every region eventually receives all local logs and

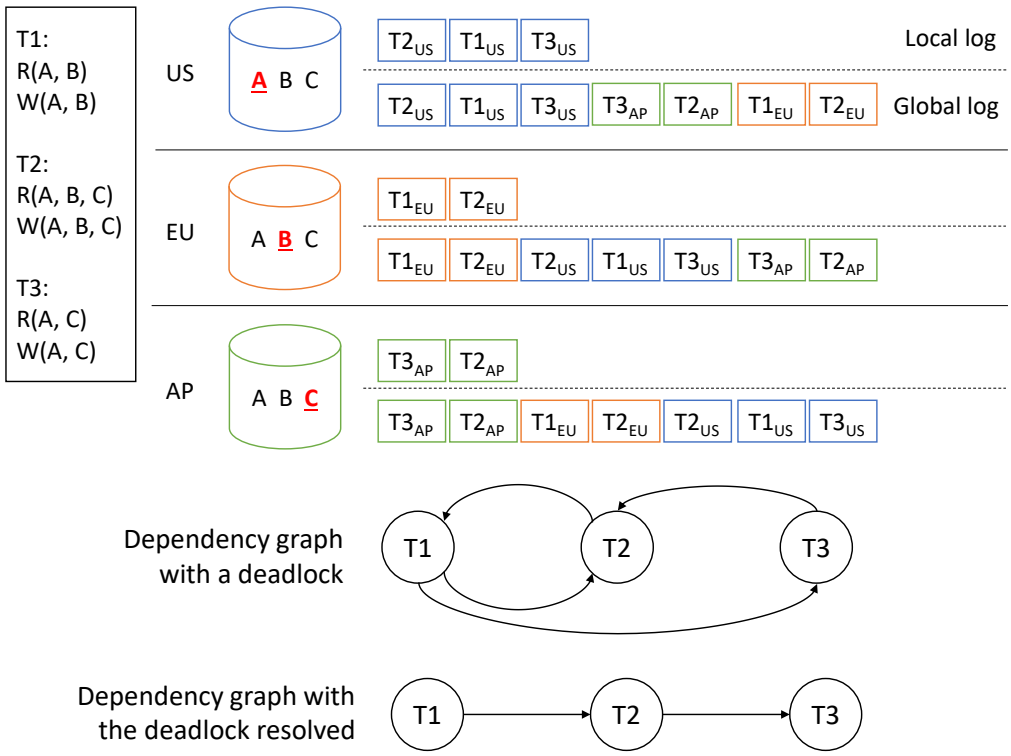


Figure 4.3: Deadlocks from multi-home transactions

constructs the dependency graph. The deadlocks manifest themselves as cycles in the dependency graph. In this case, all three transactions are in a deadlock.

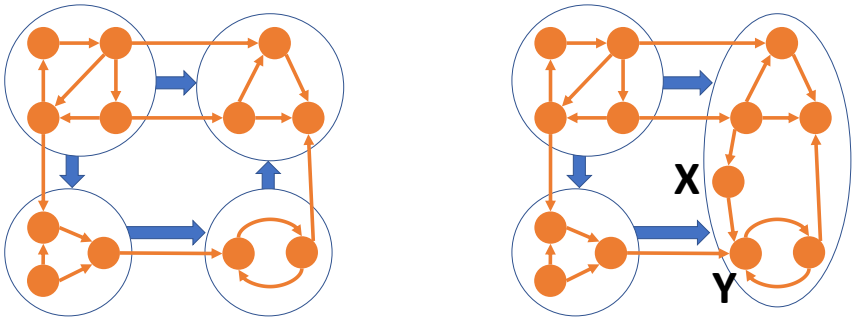
In order for the transactions to progress, the deadlocks must be eliminated, either by aborting transactions or modifying the dependencies such that the graph is free of cycles. We chose the latter approach because aborting and restarting transactions increase latency of those restarted transactions. However, a key constraint is that this modification must be deterministic: every region must independently make the same decision on how to resolve the deadlock without runtime communication across regions.

One dependency modification strategy is to serialize the transactions following the order of their IDs (see Section 4.2.1 for how IDs are generated). For example, the dependencies in the graph in Fig. 4.3 can be changed such that the processing order of the transactions is T1, T2, and T3, as shown at the bottom of the figure.

However, making this change deterministically is more complicated than it initially appears: performing the deadlock resolution as soon as a cycle is detected may cause divergence. For example, if a server in the EU region in Fig. 4.3 resolved the deadlock between T1 and T2 as soon as it saw the first four log entries in the global log, only a single edge (T1, T2) would remain between T1 and T2. When the rest of the log arrived, the edges (T1, T3) and (T3, T2) were added to the graph. This final graph is a DAG whose topological order is T1, T3, and T2, which is different from what the order would have been if the deadlock was resolved only after receiving all the log entries. Therefore, both the timing for when to run deadlock resolution along with the resolution itself must be deterministic. This is complicated by the fact that each region interleaves local log records into its global log differently and waiting too long to resolve deadlocks increases latency.

Deterministic deadlock resolution (DDR). We give an intuition for our deadlock resolution algorithm by considering its naïve version over a finite set of transactions. Each region waits until all transactions arrive then constructs a *condensation* of the dependency graph. A condensation of a directed graph $\mathcal{G}$ is formed by contracting each strongly connected components (SCC) into a super vertex, and adding a directed edge between two super vertices U and V if there is a directed edge in $\mathcal{G}$ that starts in U and ends in V (e.g. Fig. 4.4a). A condensation is a DAG since it does not have any SCC over its super vertices with a size larger than one. Therefore, we can find a topological order on the condensation. We additionally impose an order agreed upon by every region on the vertices within each super vertex (e.g. by their IDs). Determinism across regions can then be achieved by executing the transactions following both of the these orders.

In reality, it is extremely inefficient or impossible to wait for all transactions to arrive as the set of transactions may not be finite. On the other hand, running the algorithm at arbitrary time may cause the regions to see different condensations. For



(a) (b)

Figure 4.4: Example dependency graphs (in orange) and their condensations (in blue)

example, another region will observe a different condensation as shown in Fig. 4.4b if it runs the algorithm after the transaction X arrives, merging two of the SCCs together. To avoid this problem, the DDR algorithm identifies a *stable subgraph* then finds and executes the SCCs only within this subgraph. These SCCs are guaranteed to never mutate as new transactions arrive, thus ensuring convergence across the regions.

For clarity, we initially assume that the data at each region is not partitioned and will remove this assumption shortly. For every vertex corresponding a multi-home transaction T in the dependency graph, let $GP_{total}(T)$ be the total number of GraphPlacementTxns generated for T, a counter $GP(T)$ is associated with T to keep track of the number of GraphPlacementTxns of T that have been added to the graph so far. We define two types of vertices:

- A *complete vertex* T is either a single-home transaction or a multi-home transaction with $GP(T)$ equal to $GP_{total}(T)$.
- A *stable vertex* T is a *complete vertex* and there does not exist a path going from an incomplete vertex to T.

For example, before X is added to the graph in Fig. 4.4b, Y was not a complete vertex because the edge (X, Y) being missing implies at least one GraphPlacementTxn of Y was not added to the graph. As a result, any vertex that Y had a path to was not stable.

Let $\mathcal{G}(\mathcal{T}, \mathcal{E})$ be a dependency graph comprised of a vertex set $\mathcal{T}$ and an edge set $\mathcal{E}$. The *stable subgraph* of $\mathcal{G}$ is the graph $\mathcal{G}'(\mathcal{T}', \mathcal{E}')$ such that $\mathcal{T}'$ is the subset of $\mathcal{T}$ that contains all stable vertices and $\mathcal{E}'$ is the subset of $\mathcal{E}$ that contains all edges whose both incident vertices are in $\mathcal{T}'$. The stable vertices set $\mathcal{T}'$ can be found with breadth-first search, starting from the set of incomplete vertices; any traversed vertices (including the starting vertices) are marked as unstable; the remaining untraversed vertices are stable vertices.

The DDR algorithm finds all SCCs in the stable subgraph $\mathcal{G}'$; for each found SCC, it removes all its edges and adds a new simple chain of edges between the vertices found within that SCC ordered by their transaction IDs. After running this algorithm, the stable subgraph $\mathcal{G}'$ will become a DAG and, as an optimization, can be ignored in subsequent runs of the algorithm. We run this algorithm in a background thread (Algorithm 3, Line 21-25) at a configurable interval, which we set to 40 ms. This thread builds its own copy of the graph and communicates with the Scheduler via a message queue to avoid access conflict on an otherwise shared graph.

When the data is partitioned within a replica, each partition may only have a partial view of the dependency graph. Therefore, we make two modifications to the above algorithm. First, each partition will periodically broadcast its view of the graph to all other partitions (Algorithm 3, Line 8). Second, for every vertex T, let $Part_{total}(T)$ be the number of partitions participating in T and $Part(T)$ be the number of partitions participating in T that have sent their views that included T to the current server; for the vertex T to be considered complete, regardless of whether it is single-home or multi-home, it must satisfy that $Part(T)$ is equal to $Part_{total}(T)$, in addition to the conditions stated above.

4.2.3 Proof of correctness

We now prove that DETOCK achieves determinism and strict serializability. We use "component" as short for "strongly connected component". To simplify the proof, we slightly modify the DDR and transaction execution algorithms. After reordering the vertices within a component $\mathcal{C}$, the deadlock resolver adds a *virtual edge* visible only to the deadlock resolver from the last vertex to the first vertex of the series, so that the vertices in $\mathcal{C}$ continue to form a strongly connected component after reordering. The transactions are executed following the non-virtual edges. After a transaction's execution, it is marked as executed instead of being removed from the graph, and its outgoing edges become virtual edges.

Definition 4.2.2 *A region R determines a vertex T to be in a component $\mathcal{C}$ at a prefix p of the global log in R if and only if the DDR algorithm computes T to be in $\mathcal{C}$ in the stable subgraph of the graph constructed from p.*

Lemma 4.2.3 *For any two regions R_A and R_B and two conflicting transactions T_i and T_j, if R_A determines T_i and T_j to be in the same component at some prefix p_A of the global log in R_A, then if R_B determines T_i and T_j to be in some component(s) at some prefix p_B of the global log in R_B, it also determines that T_i and T_j to be in the same component.*

(By contradiction) Assume that R_B determines T_i and T_j to be in two different components $\mathcal{C}_i$ and $\mathcal{C}_j$, respectively, at p_B. In R_B, since T_i and T_j are determined to be in two different components, there does not exist a path either from T_i to T_j or from T_j to T_i in the graph constructed from p_B. Without loss of generality, we assume that the path from T_i to T_j does not exist.

In R_A, since T_i and T_j are determined to be in the same component, there exists a path ρ from T_i to T_j in the graph at p_A.

While ρ does not exist in R_B at p_B, there always exists in R_B at p_B a subpath of ρ ending in T_j (the subpath containing only T_j is one such subpath). Let T_k be the starting vertex of the longest such subpath. T_k cannot be T_i because the whole path ρ does not exists in R_B at p_B. Hence, there exists a vertex T_h that is immediately precedes T_k on ρ.

The edge (T_h, T_k) does not exist in R_B at p_B because the subpath from T_k to T_j is already the longest. Per DETOCK's protocol, R_B will eventually construct the edge (T_h, T_k) at some extension of p_B. This means T_k is an an incomplete vertex at p_B, which makes T_j an unstable vertex at p_B because there is a path from T_k to T_j. This is a contradiction because R_B cannot determine T_j to be in the component C_j at p_B if T_j is still not stable. Therefore, R_B must determine T_i and T_j to be in the same component at p_B.

Lemma 4.2.4 *If a region R determines a vertex T to be in a component $\mathcal{C}$ at some global log prefix p in R, then R determines T to be in $\mathcal{C}$ at any extension of p.*

Let $\mathcal{A}_p$ be the set of vertices each of which has a path leading to T ($\mathcal{A}_p$ contains T) in the graph constructed from p. Let p' be some extension of p.

$|\mathcal{A}_{p'}| \geq |\mathcal{A}_p|$ because we don't remove vertices. $|\mathcal{A}_{p'}| > |\mathcal{A}_p|$ only if there exists a vertex S at p' such that $S \notin \mathcal{A}_p$ and S has an outgoing edge pointing to a vertex in $\mathcal{A}_p$. An edge pointing to some vertex V can only come from a transaction preceding V in the global log, but all vertices in $\mathcal{A}_p$ are already complete because T is a stable vertex at p. Therefore, such a vertex S does not exist. Hence, $|\mathcal{A}_{p'}| = |\mathcal{A}_p|$.

Consequently, the size of $\mathcal{C}$ does not grow as the global log extends from p to p'. $\mathcal{C}$ also does not shrink because we don't remove edges and vertices as viewed by the deadlock resolver. As a result, R still determines T to be in $\mathcal{C}$ at p'.

Theorem 4.2.3 implies that two regions eventually agree on which vertices constitute a component. Theorem 4.2.4 implies that once a determination is made, it stays true

forever. Together these lemmas assert that the DDR algorithm is deterministic. Next, we show that DETOCK guarantees strict serializability.

Lemma 4.2.5 *The execution order of any two conflicting transactions T_i and T_j is the same in every region.*

It follows from lemmas 4.2.3 and 4.2.4 that eventually all regions agree on one of the following cases:

T_i and T_j are in the same component. The DDR algorithm deterministically reorders them by their IDs, and they are executed following this order in every region.

T_i and T_j are in different components. Since they conflict with each other, without loss of generality, there is a path from T_i to T_j in the dependency graph in every region. By the execution algorithm, T_i is executed before T_j, and this order is the same in every region.

Proposition 4.2.6 *Transaction schedules are strictly serializable.*

DETOCK eliminates cycles in the dependency graph using the DDR algorithm, thus its execution schedule follows the topological order of a DAG, which can be written as a serial schedule. Because of Theorem 4.2.5, all regions follow the same execution order, hence DETOCK guarantees one-copy serializability.

Furthermore, non-concurrent transactions are executed according to their temporal order in this serializable schedule: Let T_i and T_j be two conflicting transactions such that T_j is sent to DETOCK after T_i is executed and returned to a client. T_i getting executed means that the component containing T_i (which may include only T_i) is part of a stable subgraph. Therefore, T_j cannot be in the same component as T_i, thus must be executed strictly after T_i. As a result, execution of transactions in DETOCK is strictly serializable.

4.2.4 Avoiding livelock

The DDR algorithm finds and resolves **s**table SCCs that will never grow as new transactions are added to the graph. These stable SCCs correspond to deadlocks which it deterministically resolves. However, unstable SCCs also correspond to deadlocks, which the DDR algorithm cannot immediately resolve. In theory, it is possible for an SCC to grow indefinitely and never become stable, which results in livelock. Although such livelock is easy to prevent by forcing coordinators to run an admission control algorithm that temporarily holds back transactions that conflict with transactions tied up in a large unstable SCC until that SCC becomes stable, such admission control increases transaction latency, and should only be used as a last resort. Preferably, large unstable SCCs should be prevented in the first place. This is equivalent to taking measures to limit situations where deadlock is likely to occur.

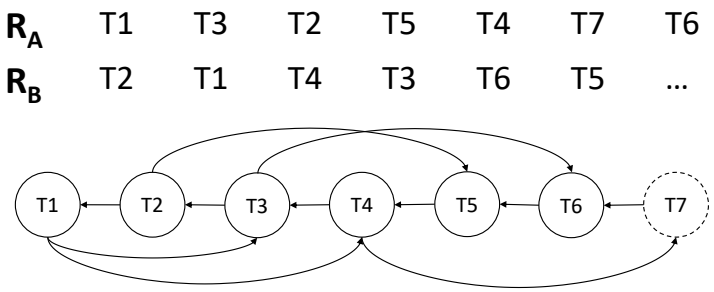


Figure 4.5: An example where an SCC can grow indefinitely.

Deadlock occurs when different regions insert multi-home transactions into their respective local logs in different orders. High network delay between two regions increases the probability of this occurring. For example, assume there are 2 regions R_A and R_B that are hundreds of milliseconds apart in terms of network delay. The whole database consists of two data items A and B, local to R_A and R_B, respectively. All transactions in this example access both of these keys and thus are multi-home transactions. The first transaction T1 starts in R_A and enters the local log of R_A almost immediately, while it takes hundreds of milliseconds for T1 to reach R_B. In the

meantime, another transaction T2 starts in R_B and enters the local log of R_B before T1 reaches R_B. It takes some time for T2 to reach R_A, but before that happens, T3 starts in R_A and enters the local log of R_A, and so on. Fig. 4.5 shows the local logs and dependency graph of up to 7 transactions that got into this scenario. At any time, the SCC cannot be resolved because the last transaction on this chain is always an incomplete transaction and has a path leading to every other transaction (T7 in Fig. 4.5).

The best way to avoid deadlock and livelock is to attempt to have multi-home transactions be inserted into every relevant region's local log in the same order. However, DETOCK's performance requirements prevents it from being able to globally order multi-home transactions before they begin, as done in other systems (such as SLOG, Fauna [50], and Calvin [123]). Instead DETOCK uses a best-effort scheme called *opportunistic ordering* that merely reduces the probability of conflicting orders of multi-home transactions (and thus deadlock), but does not eliminate it entirely.

When a transaction T first enters DETOCK, its coordinator assigns it a future (real time) timestamp based on its local clock (Algorithm 1, Line 18). Each participating region inserts T into its local log as soon as possible after its local time exceeds this timestamp (Algorithm 2, Line 4). Thus, if two transactions reach a region before their designated start times, they can be inserted into the local log at that region by this timestamp order. This is true, even if the clock at that region is not synchronized with the clocks at the regions which originally generated the timestamps of these transactions. Therefore, if these two transactions arrive **everywhere** such that their timestamps are later than the local clock at the location of their arrival, they are guaranteed to be consistently ordered everywhere.

However, if a region receives T at a local time later than the timestamp assigned to T, all it can do is to immediately insert T to its local log. Therefore, two transactions can be placed into the local log at a region out of (timestamp) order if at least one of

them arrives after its designated time. To reduce the probability of this occurring, the coordinator attempts to assign a timestamp far enough into the future so that it will arrive everywhere prior to its designated start time. To accomplish this, the future timestamp is computed by adding to the coordinator's local time the one-way delay to the farthest participating region (delay-wise) plus a small overshoot (2 ms in our implementation) (Algorithm 1, Line 18).

The one-way delay from a region R_A to another region R_B is estimated by periodically sending a message from R_A to R_B containing the sending time t_A, then R_B responses with the time offset $t_B - t_A$ where t_B is the time when R_B receives the message. The servers at R_A compute a moving average of this offset to smooth out noise and use this value as the one-way delay from R_A to R_B. This offset also incorporates the clock difference between the two regions (and thus can be negative) so this scheme does not require highly synchronized clocks. Inaccurate estimation does not affect the correctness of the system but potentially degrades its performance due to increased number of deadlocks.

4.3 Home-movement transactions

When the locality of a workload changes (e.g. a user moves to a new continent), data migration between regions is needed to keep up with access pattern changes. DETOCK carries out such data migration by using *home-movement transactions*. It performs home movement within a transaction to ensure strict serializability and avoid down time. Our description focuses on home-movement transactions that involves only one data item at a time, but it is straightforward to extend this to multiple data items.

As mentioned previously, the identifier of a data item's home region is physically stored next to the data itself. A home-movement transaction's only action is to modify this identifier. In Algorithm 1, a home-movement transaction T_{hm} has three

self-explanatory fields: *movedKey*, *oldHome*, and *newHome*. The values of *movedKey* and *newHome* are determined by the DETOCK system component (or system administrator) that decides that the data item should be located at a particular region and submits the home-movement transaction. The value of *oldHome* is retrieved from the `HomeDirectory` index at the transaction's coordinator (Algorithm 1, Line 5). Unlike ordinary transactions, a home-movement transaction does not only store the current home of a data item in the *homeInfo* map but also its soon-to-be new home (Line 6). Consequently, a home-movement transaction is always a multi-home transaction.

The home-movement transaction T_{hm} is treated exactly like an ordinary multi-home transaction from this point onward. Each region processes the transaction after receiving its two component GraphPlacementTxns, T_{hm}^{old} and T_{hm}^{new}, and then updates its `HomeDirectory` accordingly. Concurrent transactions which access the moved data item may see the old or new home location when entering the system. If they see the old location, they will be sent there and inserted into the old region's local log. If it gets inserted into that local log after T_{hm}^{old}, it will abort and restart at runtime during home validation. If it gets inserted prior to T_{hm}^{old}, it logically occurs before the data item moved, and will succeed.

According to Theorem 4.2.1, two transactions conflict not only on the key they access but also on the expected home region. If T is a transaction that sees the old location and is placed before T_{hm}^{old} but after T_{hm}^{new} in the global log, this definition prevents T from being blocked by T_{hm}^{new} because although they access the same key, T expects the key's home region to be *oldHome* whereas T_{hm}^{new} expects that to be *newHome*.

Discussion. SLOG also introduces a home-movement algorithm [109]. However, the algorithm described here is both easier to reason about and simpler to implement. Additionally, SLOG's home-movement algorithm requires storing a counter in the header of every data item, thus increasing the size of the database, while this new

algorithm does not require such a counter. The key difference that enables these advantages is that SLOG's algorithm makes the home-movement transactions single-home, whereas DETOCK's algorithm constructs them as multi-home transactions.

4.4 Implementation

We re-implemented SLOG in a new, cleaner, and more extensible codebase[2]. Building on this foundation, we developed DETOCK[3], culminating in a prototype with over 16,000 lines of C++ code in total. For communication between nodes and threads, we utilized ZeroMQ [148]. DETOCK features pluggable storage layers, defaulting to an in-memory key-value store. Transactions are handled as stored procedures that perform read and write operations on a specified set of keys.

4.5 Evaluation

The goal of DETOCK is to achieve high throughput and low latency for strictly serializable transactions over a geo-replicated and geo-partitioned database. Hence, we compare DETOCK to four other systems that also support globally distributed

[2]Available at https://github.com/umd-dslam/SLOG
[3]Available at https://github.com/umd-dslam/Detock

Table 4.1: Round-trip time for all pairs of regions (ms)

	apse2	apse1	apne2	apne1	euw2	euw1	usw2*	usw1*	use2
use1	197	211	173	148	75	67	66	61	12
use2	187	197	160	132	85	77	52	50	
usw1*	137	169	134	107	145	136	20		
usw2*	139	174	124	95	128	127			
euw1	254	183	229	202	11				
euw2	263	171	236	209					
apne1	128	71	32						
apne2	148	72							
apse1	91								

* Regions used only in the CockroachDB experiment.

transactions: Calvin [123], SLOG [109], Janus [88], and CockroachDB [118]. To reduce performance artifacts that are unrelated to the architectural designs discussed in this chapter, we re-implemented Calvin, SLOG, and Janus inside the DETOCK codebase so that all four systems can use the same storage layer, communication library, local consensus code, and logging infrastructure. Unlike DETOCK, Calvin globally orders all transactions, and SLOG globally orders all multi-home transactions. The SLOG paper discusses two ways to do this: (1) sending them all to the same region/ordering service and (2) performing global consensus via Paxos or Raft. Option (2) increases the latency of every multi-home transaction by at least the latency of the global consensus protocol (hundreds of milliseconds for a truly global deployment). Option (1) only increases the latency of multi-home transactions that initiate far from the ordering service. We experiment with both versions in Section 4.5.2, but since option (1) yields better latency, we use it for both Calvin and SLOG in the other experimental sections, in order to present their latency in the best possible light, even though it is less robust to region failure than option (2). Janus generalizes the EPaxos protocol [86] to process distributed transactions, which (similar to DETOCK) includes a reordering technique over a dependency graph and execution of transactions across all replicas and shards deterministically following the graph order. However, unlike DETOCK's asynchronous protocol, Janus synchronously replicates data to every region so it needs at least one WAN round-trip to all regions to commit.

We choose CockroachDB as another comparison point because it has support for state-of-the-art geo-partitioning that uses a non-deterministic approach. CockroachDB inherits many of its architectural principles from Spanner [31]. Both CockroachDB and Spanner are widely used; however CockroachDB is the better comparison point since it is available as independent downloadable code and can be deployed on the same cluster as our other experimental systems, and supports geo-partitioning. Nonetheless, it is far more production-ready than the research prototype of DETOCK, uses a more robust

and fully-featured storage layer, and any raw performance numbers with DETOCK would be oranges to apples. Therefore, we only report relative measurements and observe the performance trends of each system.

Unless stated otherwise, we ran our experiments on Amazon EC2 using r5.4xlarge instances. Each machine has 16 vCPU and 128GB of memory. We deployed the systems over 8 AWS regions: us-east-1 (N. Virginia), us-east-2 (Ohio), eu-west-1 (Ireland), eu-west-2 (London), ap-northeast-1 (Tokyo), ap-northeast-2 (Seoul), ap-southeast-1 (Singapore), and ap-southeast-2 (Sydney). Table 4.1 contains the round-trip time for every pair of regions. In each region, a replica of the database is partitioned across 4 machines.

The clients generating the workloads were deployed on separate machines spread evenly across all regions, and had enough capacity to avoid being bottlenecks. Each client thread issued one transaction at a time. Transactions are either single-home (SH) or multi-home (MH); separately, as an unrelated consideration, they can be either single-partition (SP) or multi-partition (MP). SH transactions access data in the region closest to the client that generates it. MH transactions access data from two regions. Calvin and Janus do not assign home regions to data items, and do not support geo-partitioning so their transactions can only be SP or MP. Each client weights the regions following a Zipfian distribution such that regions closer to the client are more likely to be selected for a MH transaction. We vary the percentage of MH and MP transactions.

4.5.1 Microbenchmark experiments

In our first set of experiments, we use a version of the Yahoo! Cloud Serving Benchmark (YCSB) [30] adapted for transactions. The data consists of a single table containing a billion rows and two columns: a 64-bit integer key and a value consisting of 100 random bytes. Identically to previous work running experiments on this same dataset [109],

contention of the workload is varied by dividing the table into "hot records" and "cold records". A transaction performed read-modify-write on 2 hot records and 8 cold records; all records were uniformly selected at random. Contention is varied by changing the size of the hot record set. We define HOT to be the reciprocal of the size of the hot re cord set per partition. Thus, contention increases with the value of HOT. Our initial set of experiments places all machines in the us-east-2 (Ohio) region, and uses tc [120] to simulate the network round-trip time of the 8-region deployment with symmetric network paths and a jitter uniformly distributed within 1 ms. This will allow us to directly vary network conditions in Section 4.5.1. In Section 4.5.2 we remove the simulation and run over the real full 8-region deployment.

Throughput

Fig. 4.6 shows the peak throughput of Calvin, Janus, SLOG and DETOCK, with and without opportunistic ordering. We varied the % MP and % MH parameters at two HOT settings corresponding to a low contention workload (HOT = 0.0001) and a high contention workload (HOT = 0.01).

Calvin and Janus do not distinguish between SH and MH transactions (since they do not support geo-partitioning); thus their throughput stays constant as % MH is varied. In contrast, DETOCK and SLOG are able to benefit from the presence of SH transactions in workload by processing them at different regions in parallel, while Calvin and Janus have to order all transactions within a single global log. Therefore, for workloads which geo-partition well (e.g. 15% or fewer MH transactions), DETOCK and SLOG significantly outperform Calvin and Janus. However, as MH% increases past 30%, the geo-partitioning advantage of DETOCK and SLOG disappears and all systems perform similarly. At extremely high levels of MH transactions, they even perform slightly worse than Calvin, since they incur additional overhead to process MH transactions (i.e. dividing the transaction into its region-local components, and

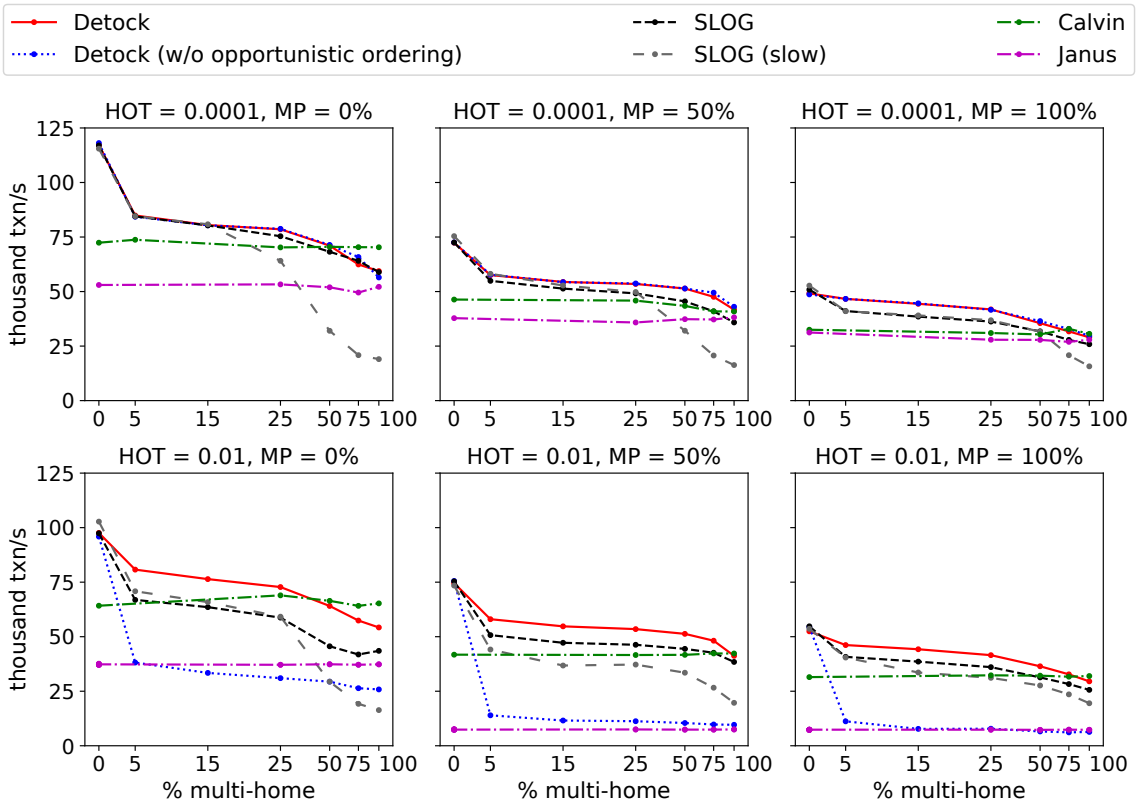


Figure 4.6: Microbenchmark throughput

inserting in the local log of each region). However, this lower throughput relative to Calvin only occurs when the MP% is 0. When MP transactions are common, all systems have to pay overhead processing the transaction across the different local machines that own the partitions of data accessed. Therefore, Calvin only outperforms DETOCK in the scenario of high MH% and low MP% — an unlikely scenario since if a workload is partitionable, it is generally locally geo-partitionable as well.

Janus requires traversal of the dependency graph, including communication with other shards for missing information, on the critical path of every transaction to determine when it is ready to be executed. This overhead causes Janus to perform worse than other systems. Conversely, DETOCK traverses the dependency graph and communicates with other partitions in a background thread that wakes up periodically, thus its cost is amortized across the periodic runs. The throughput of Janus drops

further under high contention, especially when there are MP transactions, because the graph grows faster and more cross-shard messages are needed.

Under low contention (top row), the best versions of DETOCK and SLOG have similar throughput. However, under high contention (bottom row), DETOCK outperforms SLOG when there are MH transactions in the workload. This is because SLOG must globally order all MH transactions. Different regions involved in the transaction find out the order (and then insert the transaction into their local log) at different points in time. The closer they are to the region that determines the order, the faster they get started with the transaction. However, a transaction cannot release locks until they receive local logs from all regions involved in the transaction, even remote regions. The slowest transactions in SLOG therefore must hold locks for longer than the slowest transactions in DETOCK. Under low contention, this does not affect performance. But under high contention, this longer hold time reduces throughput. In contrast, DETOCK uses opportunistic ordering to insert the GraphPlacementTxn to the local log at roughly the same time, thus distributing the blocking time evenly across the regions.

SLOG's performance depends on resources allocated for its ordering service. To show this, we ran a version of SLOG where we cut down the number of threads used for deserializing and batching the transactions in the ordering service from 8 to 1. The throughput of this version, shown as SLOG (slow) in Fig. 4.6, quickly drops as the amount of MH transactions increases. DETOCK is not bounded by this constraint because it does not need an ordering service.

Surprisingly, even at large numbers of multi-home transactions, DETOCK's performance is almost unchanged relative to its performance at low contention. Such independence from performance degradation for strictly serializable multi-region transactions at extremely high contention is rare amongst current state-of-the-art systems and is an important advantage of DETOCK's approach. The comparison to

CockroachDB's state-of-the-art geo-partitioning system in Section 4.5.4 will further highlight the DETOCK's exceptional resilience to high contention workloads.

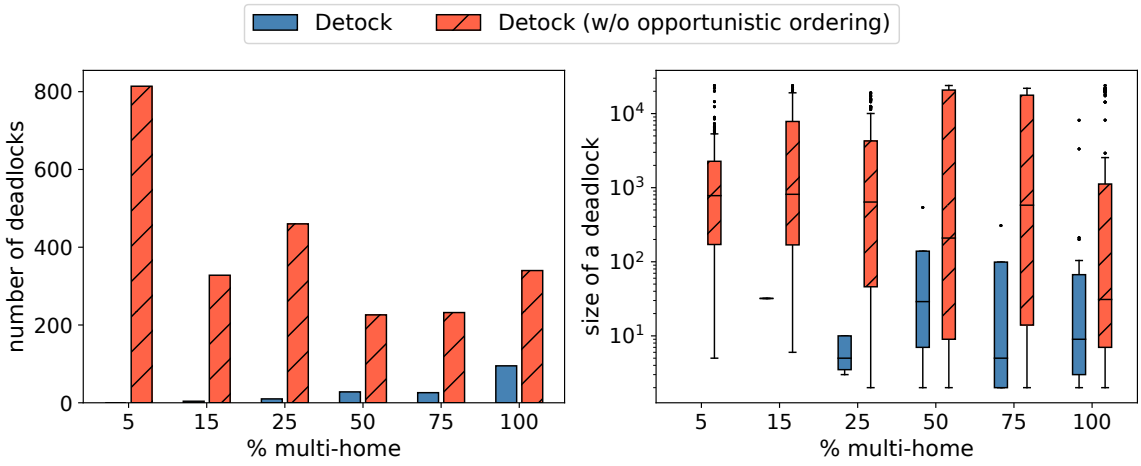


Figure 4.7: Impact of opportunistic ordering on deadlock number and size

To understand the drop of throughput of the DETOCK version without opportunistic ordering, we plotted the number and size of deadlocks (SCCs) of the two DETOCK versions for HOT = 0.01 and MP = 100% in Fig. 4.7. The reason for the performance drop is thus due to the growth of the number and size of deadlocks. When the contention is high, the probability that new incomplete dependencies emerging while a deadlock is forming increases, preventing the DDR algorithm to immediately resolve the deadlock.

Latency

We measured the end-to-end latency using a smaller number of clients to avoid including queuing time. Fig. 4.8 presents the p50 and p99 latency at 100% MP. Under low contention, SLOG and DETOCK[4] achieve low latency for SH transactions as expected. Meanwhile, Calvin must globally order **all** transactions so it results in

[4]The latency difference between DETOCK with and without opportunistic ordering is the amount of the overshoot (2ms). All other delays caused by opportunistic ordering is overlapped with transaction processing and do not cause a latency increase. To avoid cluttering the graph, we only show DETOCK **with** opportunistic ordering.

an order of magnitude worse latency. Under high contention, it becomes more likely for SH transactions to conflict with the longer-running MH transactions, so SLOG and DETOCK have higher p99 latency for SH transactions in the presence of MH transactions. However, the impact of this on DETOCK is much lower than SLOG.

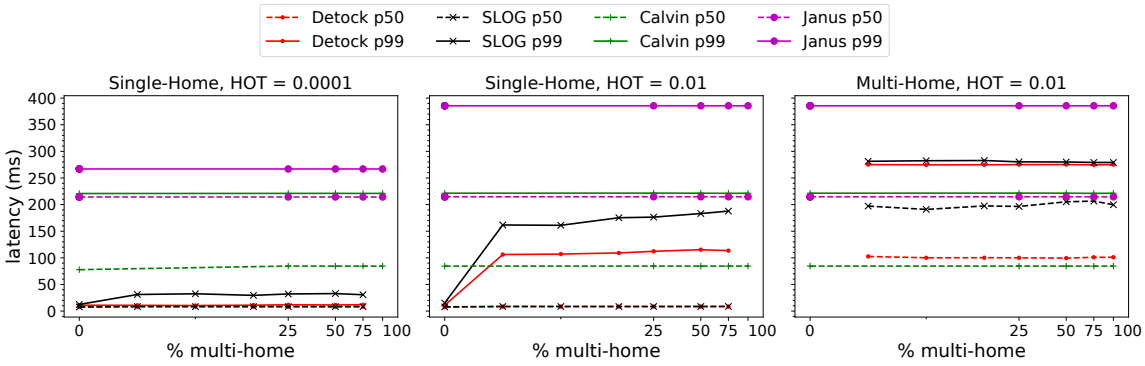


Figure 4.8: Microbenchmark latency (MP = 100%)

For MH transactions, SLOG has a latency higher than Calvin's and DETOCK's because every MH transaction in SLOG needs (1) a round-trip to the ordering region and (2) additional communication to exchange the local logs across regions. Calvin also must pay cost (1) but avoids cost (2). DETOCK must pay cost (2) but avoids cost (1). Therefore they have similar latency at p50. However, cost (2) has a longer tail latency when locks are held during this communication; therefore, Calvin achieves better p99 latency for MH transactions.

Janus has the highest latency because its fast quorums contain all replicas, thus every transaction coordinator generally has to wait for the response from the slowest region.

Performance under different network conditions

The effectiveness of opportunistic ordering and consequently the overall performance of DETOCK is affected by the accuracy of its estimation of the one-way delay between two regions, and irregular network conditions may affect such estimation. Therefore, we

now study the effect of asymmetric network delay and jitter on DETOCK performance. In the following experiments, we ran a workload with HOT = 0.01 (high contention), MP = 100% and MH = 10%.

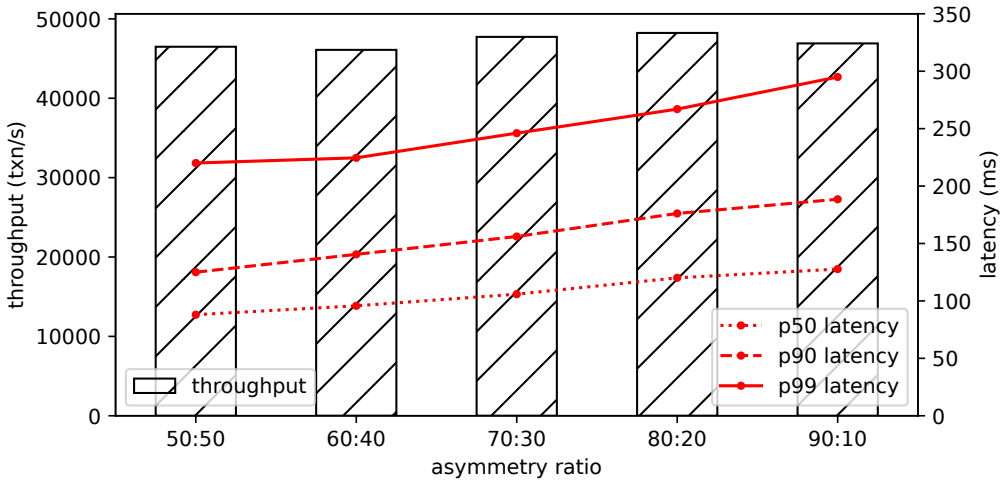
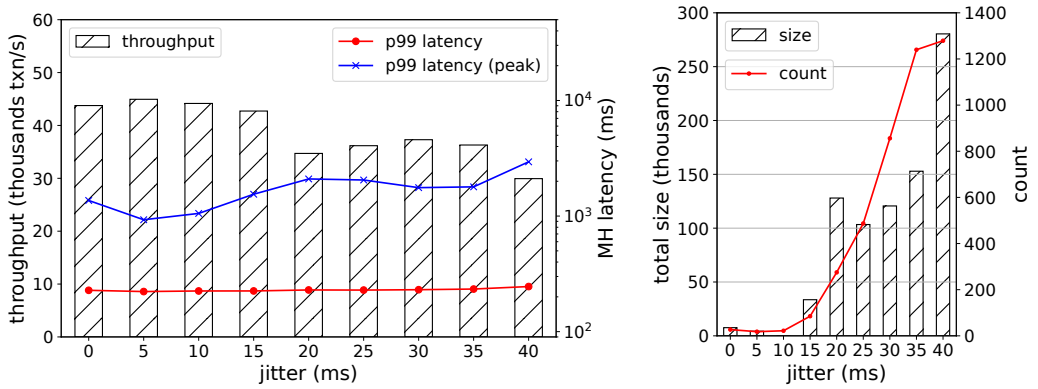


Figure 4.9: DETOCK's performance under asymmetric delay

A network path is asymmetric if the forward and backward one-way delays differ. We varied the ratio between the one-way delays on the same network path. This ratio was applied to all paths of every pair of servers across two different regions, with the direction of asymmetry randomly chosen. Fig. 4.9 shows that delay asymmetry does not affect throughput. This is because the opportunistic ordering scheme constantly monitors the one-way latency between regions and adjusts its predictions accordingly. Accurate one-way predictions are sufficient to avoid deadlock.

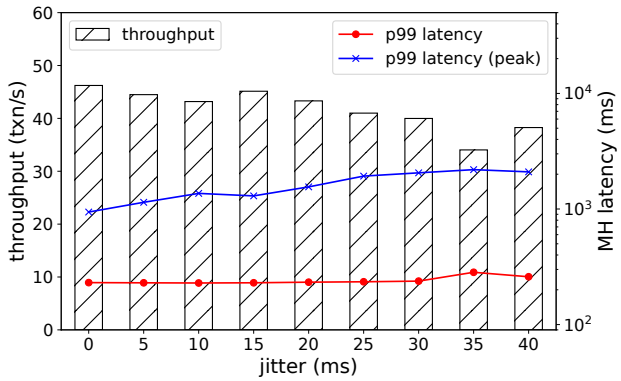
In contrast to throughput, latency increases as the ratio becomes more extreme. This is because in an asymmetric network, the region with the largest one-way delay from the coordinator might not be the one with the largest round-trip time and vice-versa. Therefore, opportunistic ordering scheme might cause the farthest region from the coordinator to hold off the transactions as if it is a closer region, increasing the overall latency of the transaction.

Nonetheless, the impact on overall latency is insignificant except at extreme

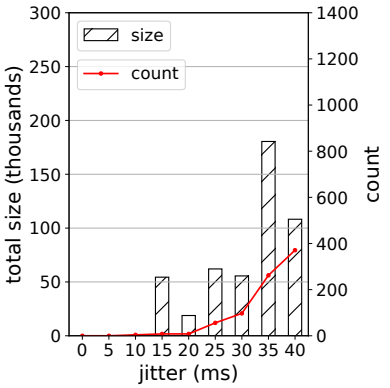


(a) Throughput and latency (2 ms)

(b) Deadlocks (2 ms)



(c) Throughput and latency (10 ms)



(d) Deadlocks (10 ms)

Figure 4.10: Network delay jitter experiments (numbers in parentheses are the opportunistic ordering overshoots)

asymmetries, yet studies have shown that 90% of the measured one-way delay on the Internet are within 40%–60% of the round-trip time [95]. Furthermore, consistently severe asymmetry is unlikely in real-world data center environments [137].

Network jitter is variance in network delay. We simulated different uniform jitter values in all inter-region paths. Fig. 4.10a shows peak throughput, p99 latency at peak throughput (blue line), and p99 latency at a lower load (red line). Increase in jitter impacts peak throughput more than latency at an unsaturated load. Fig. 4.10b shows the reason for this result: at 15ms or more jitter, opportunistic ordering's delay estimations become inaccurate, and it becomes increasingly ineffective at preventing deadlocks, which reduces throughput. DETOCK deployments with high network jitter

need higher overshoots. Fig. 4.10b showed that, with the default overshoot of 2 ms, DETOCK is robust to jitter of up to 15 ms. Fig. 4.10c and 4.10d show that increasing to a 10 ms overshoot significantly reduces the number of deadlocks, thereby raising throughput.

4.5.2 TPC-C

In this section, we evaluate DETOCK on the TPC-C benchmark [125] that is designed based on the activities of a wholesale supplier with 9 tables and 5 types of transactions. TPC-C data is typically partitioned by the warehouse table and we follow this partitioning by assigning different warehouses across the eight physical regions in our deployment. We initialized the database with 1200 warehouses and 10 districts per warehouse. We followed the specification for the transaction mix ratio, displayed in Table 4.2. However, transactions whose access set are dependent on a read were modified to remove these dependencies, since the DETOCK codebase does not currently support dependent transactions. For example, payment transactions select customers only by their IDs (instead of combination of IDs and last names). Some new-order and payment transactions may access "remote" warehouses in addition to their default warehouses. The specification only requires a remote warehouse to be any warehouse other than the default one. However, we redefine a remote warehouse to specifically be a warehouse that resides in a remote region; hence, these transactions become multi-home transactions.

Table 4.2: TPC-C transactions mix ratio

	`NewOrder`	`Payment`	`OrderStatus`	`Delivery`	`StockLevel`
SH	40.7%	42.3%	4.1%	4.1%	4.0%
MH	4.4%	0.4%	0.0%	0.0%	0.0%
Total	45.1%	42.7%	4.1%	4.1%	4.0%

We ran the TPC-C workload while increasing the number of clients until reaching peak throughput and plotted the p50 and p99 latency at different throughputs in Fig. 4.11. DETOCK and SLOG have the same median latency because the majority of transactions in the TPC-C workload are single-home. Calvin and Janus have much higher latency since every transaction has to be globally ordered. DETOCK's 99% latency is 66ms lower than SLOG's because of its ability to avoid the global ordering step for multi-home transactions which constitute 4.8% of the workload (see above, Section 4.5.1).

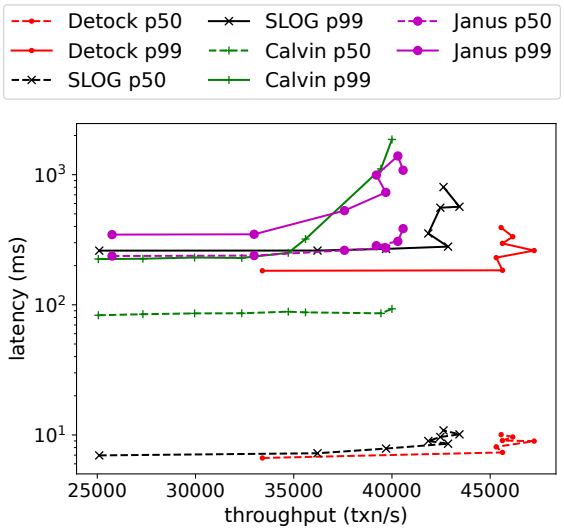


Figure 4.11: Throughput vs. latency with increasing number of clients in the TPC-C benchmark

DETOCK reaches a higher peak throughput than SLOG, Calvin, and Janus do for the same reasons discussed in Section 4.5.1.

To further explore the advantage of DETOCK for MH transactions, we plotted in Fig. 4.12 the CDF of SH and MH transaction latency in every region. Note that the x-axis is log scaled. In SLOG, the ordering service was in us-east-2. Therefore, the farther a region is from us-east-2 (US East Coast), the more benefit is accrued from DETOCK's ability to serve MH transactions without making a round-trip to the ordering service. This results in many transactions having a factor of 5 better latency

than SLOG's. Us-east-1 and us-east-2 also have improvement in their transaction latency because the ordering service incurs queuing and processing delays, which are not present in DETOCK. The Calvin version in this experiment uses one region us-east-2 for ordering, thus the latency of transactions increases as they originate farther away from the ordering region. On the other hand, every transaction in Janus generally has to wait for responses from all regions for its fast quorum, hence every region experiences high latency.

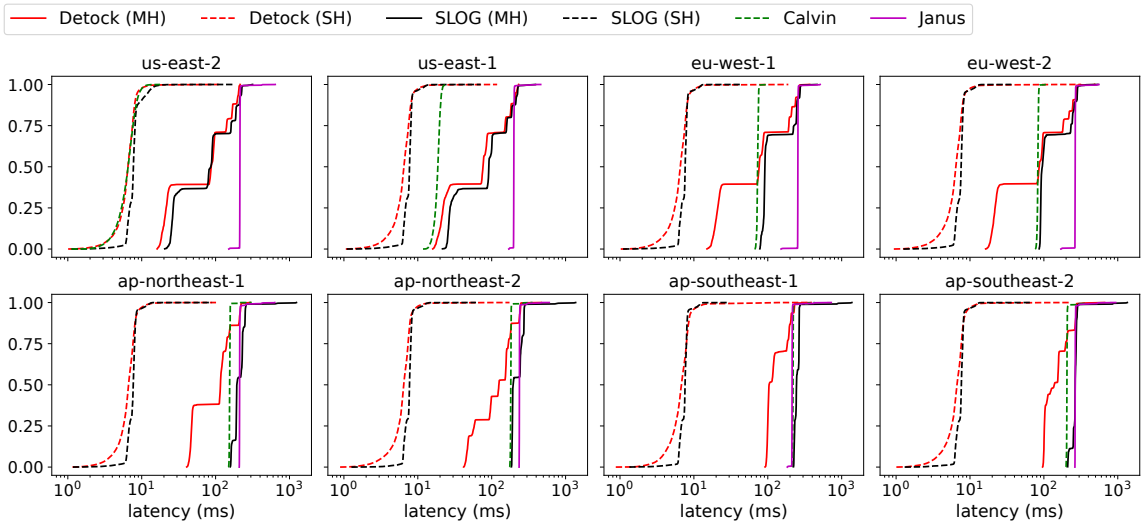


Figure 4.12: CDF of latency per region in the TPC-C benchmark

4.5.3 Scalability

We evaluate the scalability of DETOCK by running the microbenchmark as the number of machines per region increases from 3 to 21. Fig. 4.13 shows the results of DETOCK in comparison to SLOG under different settings of the parameters HOT, % MP, and % MH.

When MP and MH are 0, SLOG scales better than DETOCK due to the overhead of the background thread in DETOCK which periodically scans the dependency graph for deadlocks. This overhead can be mitigated by dynamically adjusting the activity of the background thread based on the frequency of deadlocks. The cost of dependency graph

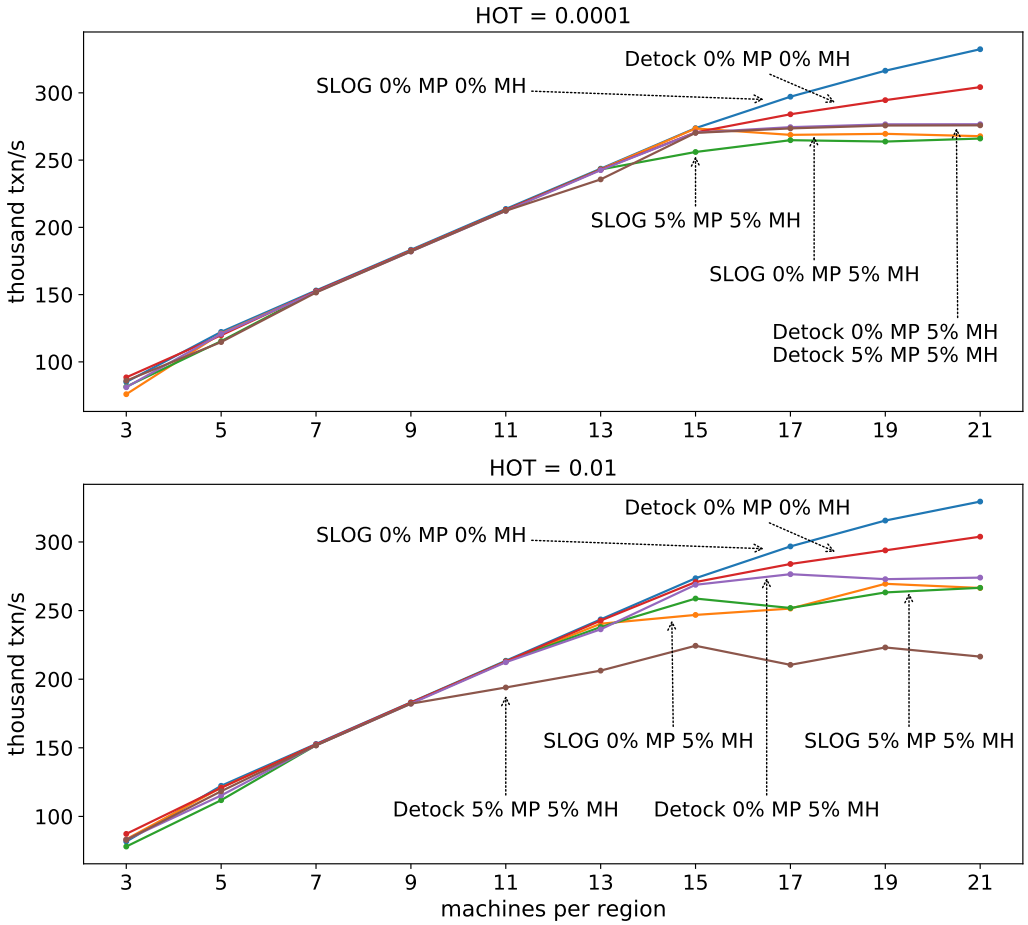


Figure 4.13: Scalability of DETOCK and SLOG

management in DETOCK grows under high contention when there are MH transactions in the workload and even more so when there are also MP transactions. Thus, DETOCK reaches scalability limitations earlier in the lower graph where contention is high. Figure 4.14 shows the reason for this is that the unstable part of the graph takes longer to be resolved and thus grows large in the presence of MP transactions, as each partition only has a partial view of the graph and needs to wait for more information from other partitions to proceed. Either way, when there are multi-home transactions in the workload, the throughput of both DETOCK and SLOG cease to increase after 15 machines per region. This is because multi-home transactions generate extra GraphPlacementTxns in DETOCK and LockOnlyTxns in SLOG, the routing of which becomes more complex as the cluster size increases. An improved routing layer would

increase the scalability of both systems.

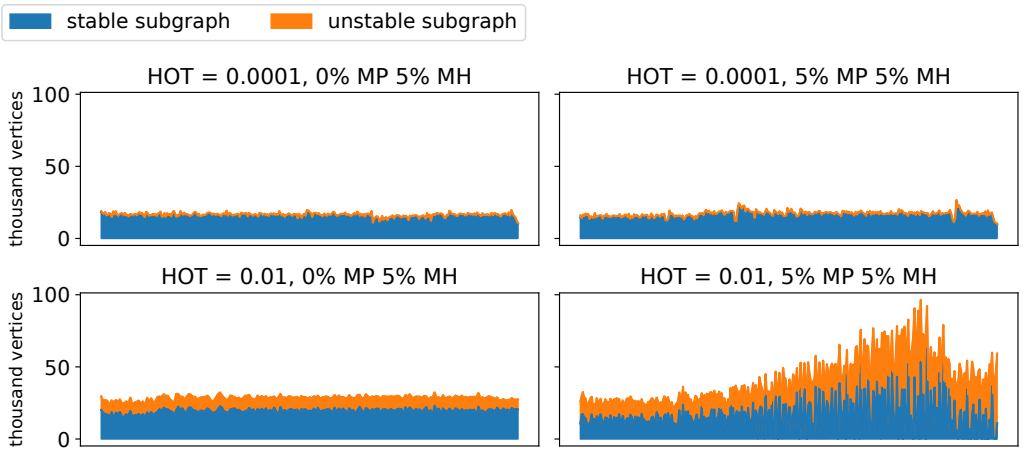


Figure 4.14: Graph size over time at 19 machines per region

4.5.4 Comparison to CockroachDB

CockroachDB is a distributed transactional database system that allows users to control the locality of individual rows, and thus compares directly with DETOCK. Although, CockroachDB does not guarantee strict serializability because it is susceptible to the causal reverse anomaly [63], it guarantees strict serializability for the vast majority of practical workloads and its architecture is based on Spanner which does guarantee strict serializability for all workloads. As discussed above, the absolute performance numbers between DETOCK and CockroachDB are incomparable because the two systems come from separate codebases. Nonetheless, information about the consequences of the architectural differences can still be gleaned from their relative performance trends.

We deployed the two systems in 6 regions: us-east-1 (N. Virginia), us-east-2 (Ohio), us-west-1 (N. California), us-west-2 (Oregon), eu-west-1 (Ireland), and eu-west-2 (London); the inter-region latency is included in Table 4.1. Each region had 3 c5.4xlarge EC2 instances (16 vCPU and 32GB memory), as recommended by CockroachDB [102]. We used CockroachDB v21.1, which was the latest version when we ran the experiment. To eliminate as many differences between the two systems as

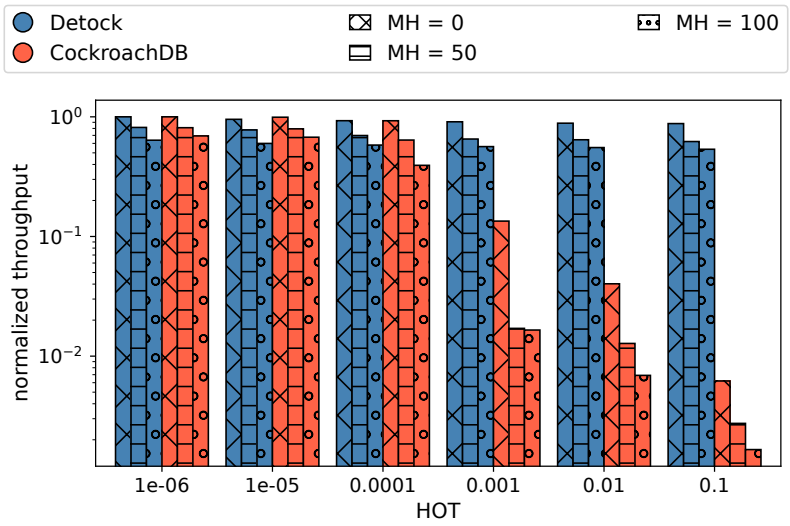


Figure 4.15: Normalized throughput

possible, we configured CockroachDB such that it used the in-memory storage engine, had only one copy per replica within a region, and parsed the SQL queries only once using prepared statements. We sent every transaction in one shot, with automatic retry turned off so that we could collect abort information. We ran the YCSB workload while varying the % MH and HOT parameters. Since CockroachDB distributed data uniformly across servers in each region, most transactions are multi-partition. Therefore, we compared against 100% MP for DETOCK.

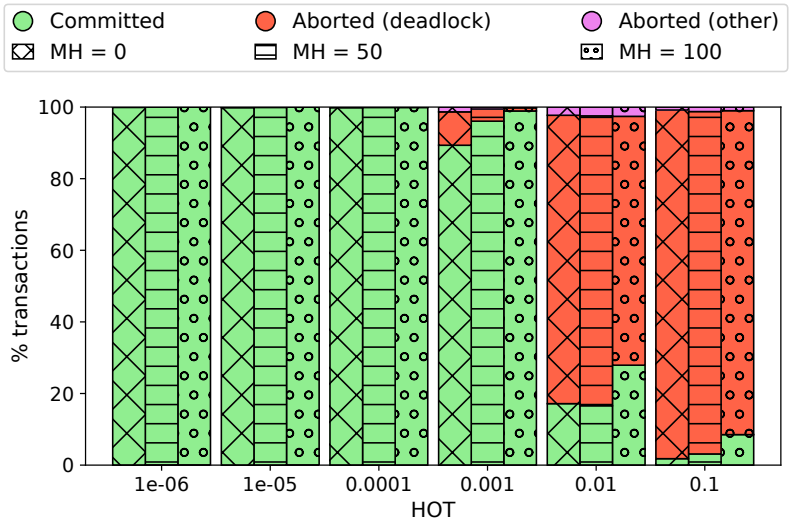


Figure 4.16: % committed/aborted of CockroachDB

The results of this experiment is normalized in Fig. 4.15 such that the throughput of DETOCK and CockroachDB at different contention levels is relative to their throughput at the lowest contention level. CockroachDB's throughput plunges at high contention, and even more at high MH%. In the worst case, the throughput of CockroachDB drops to less than 1% relative to its throughput at the lowest contention. Conversely, DETOCK's throughput decreases more gradually and slowly. It is able to retain at least 76% of the original throughput at the highest contention level.

CockroachDB partitions the database into multiple consensus groups. Its geo-partitioning feature places each group within a single region so that nearby reads and writes have low latency. CockroachDB uses a form of locking to handle write-write conflicts and thus is susceptible to deadlocks. It breaks a deadlock by randomly aborting one of the transactions. Additionally, its use of two-phase commit exacerbates the time a transaction needs to hold locks. Fig. 4.16 shows the percentage of CockroachDB's transactions that are committed, aborted due to deadlocks, and aborted due to other reasons. When contention increases, CockroachDB aborts more transactions because of deadlocks, causing wasted work. In contrast, DETOCK does not abort transactions due to deadlocks.

4.6 Conclusion

While the related work described above must trade off consistency for latency, DETOCK is able to completely side-step this trade-off for geographically partitionable workloads. Furthermore, even for non-partitionable workloads, DETOCK is able to process strictly-serializable multi-partition transactions with single round-trip latency and high throughput. Even under extremely high contention we observed near-zero performance degradation, and orders of magnitude better throughput robustness than CockroachDB.

Chapter 5

Study on assumptions of modern transactional database systems

Inspired by the pioneering work of System R [7] and Ingres [116] in the 1970s, many database systems present an interactive interface that allows a human terminal operator or application to execute commands in multiple round-trips to the server within a single transaction. Typically, the client initiates a transaction by sending a `BEGIN` command to the server, followed by multiple data access commands. The transaction concludes with either a `COMMIT` or `ABORT` command to commit or abort the transaction.

However, Stonebraker et al. [117] argued that since the landscape of applications and workloads has significantly transformed since the 1970s, it is more beneficial to develop specialized systems dedicated to transaction processing, separate from those emphasizing analytics. They posit that most modern transactional workloads do not need the flexibility of interactive transactions. Instead, a more restrictive interface, where each transaction is sent to the server in one or a few round-trips, can be exploited to process transactions more efficiently. Specifically, this new interface eliminates stalling on the client side, reduces the overhead of client-server communication, and potentially opens up the design space for optimizing concurrency control protocols.

Since then, many OLTP systems adopting this design decision have emerged. Some systems completely eschew interactive transactions, supporting only transactions executed in one round of communication between the client and the server (henceforth referred to as **one-shot transactions**). Examples of such systems include H-Store [61], Granola [32], Calvin [123], Silo [127], BOHM [46], Janus [88], PWV [47], STAR [81], Ocean vista [49], Chiller [147], Tempo [43], DynamoDB [58], and more [5], [26], [39], [56], [59], [60], [69], [75], [79], [80], [91], [99], [103], [104], [108]–[110]. Other systems still support interactive transactions but are optimized for those with one or a few round-trips, such as Carousel [138], Natto [139], and NCC [78].

In addition, there are OLTP systems operating under the premise that the complete read/write set of every transaction is known prior to scheduling and executing any part of the transaction. We refer to these system as **upfront read/write set-based systems** throughout this chapter. Such *a priori* knowledge about the read/write set can be leveraged to produce more efficient execution schedules, such as those that are deadlock-free. The read/write sets can be declared explicitly or extracted using static analysis [59]. However, these read/write set inference methods cannot be applied to *dependent transactions* [122], for which another method called Optimistic Lock Location Prediction (OLLP) [111], [123] is typically employed to optimistically determine the read/write set.

Upfront read/write set-based systems largely originate from research on *deterministic database systems* (Section 2.3), examples of which are Calvin [123], VLL [110], BOHM [46], PWV [47], SLOG [109], Q-Store [103], Caracal [104], Detock [91], and Caerus [56]. Other systems adopting this assumption include Janus [88], Carousel [138], Ocean Vista [49], Strife [99], Tempo [43], and Natto [139].

When making either of these assumptions, both of which are employed by our work on Detock in Chapter 4, the OLTP systems trade some transaction expressivity for performance gains. Under the first assumption, applications must tolerate a lack of

interactive transactions. Under the second, while general transactions can still be supported via OLLP, optimal performance is best achieved when the read/write set is readily available [111]. Nevertheless, the justifications for these trade-offs have not been supported with empirical evidence, leaving the extent to which they are applied to real-world applications unclear.

To fill this gap, we conduct a comprehensive study of 108 open-sourced applications, collectively containing more than 30,000 transactions, to investigate the prevalence of these assumptions. We focus on highly popular applications, measured by GitHub stars, built with Django (Python) and TypeORM (TypeScript), which are among the most popular ORM frameworks of their respective language. Previous studies on real-world transactions [10], [119] have also recognized the widespread adoption of ORM frameworks and targeted ORM-backed applications to gain insights into transaction usage in general. Moreover, these frameworks expose database operations as small well-documented sets of methods, enabling us to easily automate their discovery within large codebases.

We aim to answer the following questions.

1. Is it reasonable to assume that the read/write set of real-world transactions can be known in advance?
2. How common are dependent transactions?
3. Is it reasonable to assume that modern transactions are rarely interactive?
4. How common are interactive transactions?
5. How hard is it to modify existing interactive transactions to remove their interactivity?

In order to do so, we create a dataset from the applications by annotating their database operations based on definitions related to *read/write set inferability* and

transaction interactivity. Specifically, we propose four different methods to define the read/write set, each with varying trade-offs regarding the granularity of conflict detection among transactions and the ability to extract the read/write set statically from transaction code without resorting to a *fallback mechanism*, as detailed in Section 5.3.1. We quantify the latter aspect by analyzing each transaction and annotating the properties that necessitate fallbacks. Additionally, we identify whether a transaction is interactive or not and further categorize interactive transactions into those that can be converted to one-shot transactions and those that cannot due to the presence of *external operations*, described in Section 5.4.2.

The results of our study show that for 90% of applications, at least 58% of their transactions have read/write sets that can be inferred in advance when using existing upfront read/write set-based systems. This means that at most 42% of transactions in these applications require some fallback mechanism to determine the read/write set. However, the data suggests that these fallback mechanisms can be very lightweight, potentially resulting in low overhead. On the other hand, we find that OLTP systems that do not support interactive transactions can still cater to a wide range of applications—60% of the applications in this study. Furthermore, among applications that include interactive transactions, each application has an average of 9.4% of its transactions being interactive and almost all of them can be converted into one-shot transactions with minimal modifications. More importantly, our study highlights research opportunities to further enhance systems relying on these assumptions.

5.1 Object-Relational Mapping

Web applications are typically written in object-oriented languages, where data is modeled as a collection of objects interacting with each other. In contrast, relational

database systems store data in structured tables, which are accessed and manipulated using SQL. To bridge the gap between these two paradigms, developers can employ an ORM framework.

ORM frameworks automatically translate object-oriented concepts into SQL queries and map the retrieved data back to corresponding objects in memory. This approach enables developers to write both data definition and data manipulation logic in the same language as the application itself. In our study, we focus on applications built with two popular ORM frameworks: Django (Python) and TypeORM (TypeScript). These frameworks offer similar functionalities and are widely used within their respective ecosystems.

When using an ORM, developers define special classes called *models*[1], each of which is typically mapped to a database relation. The fields within these classes correspond to the relations' attributes. ORM frameworks also provide mechanisms for defining relationships (e.g. one-to-one) between fields of different models. Instead of writing raw SQL queries, developers interact with the underlying data by calling methods on objects instantiated from these classes. Some methods manipulate in-memory data on the client side, while others generate and submit SQL queries to the database server. We refer to a query submission as an **operation**. Additionally, ORMs offer APIs to group multiple operations into a single interactive transaction. Outside of interactive transactions, an operation itself run within a transaction containing a single statement (i.e. one-shot transaction). Throughout the rest of this chapter, we use the term **interactive transaction** specifically for transactions constructed via the ORM's transaction API, while using the term **transaction** to refer to either an interactive transaction or an operation that is not part of an interactive transaction.

Fig. 5.1 illustrates an example document management application using the Django ORM framework. This application includes three models: `User`, `Document`, and

[1]Referred to as "entities" in TypeORM

User

id	email
1	john@university.edu
2	jane@company.com
…	…

Document

type	create_date	path	owner
paper	06/19/2024	/doc10	1
thesis	08/15/2024	/doc20	1
…	…	…	…

PendingEmailChange

activation_key	user_id	new_email
ABCDEFGHIJK	1	john@email.com
…	…	…


```
def find_documents(type, date_range, domain=None):
  if domain is None:
    return Document.filter(type=type, create_date__range=date_range)
  else:
    return Document.filter(type=type, create_date__range=date_range,
                           owner__email__endswith=domain)

@atomic
def create_document(email, type, root_dir, name, content):
  user = User.get(email=email)
  path = root_dir / user.id / name
  if not path.exists():
    Document.create(type=type, create_date=today(), path=path, owner=user.id)
    write_file(path, content)

@atomic
def confirm_email_change(key):
  pec = PendingEmailChange.get(activation_key=key)
  if User.filter(email=pec.new_email).exists():
    rollback()
    return
  user = User.get(id=pec.user_id).update(email=pec.new_email)
  pec.delete()
```

Figure 5.1: An example of a document management application using Django (API simplified for brevity).

`PendingEmailChange`. The `id` attribute of the `User` model has a one-to-many relationship with both the `owner` attribute of the `Document` model and the `user_id` attribute of the `PendingEmailChange` model.

The application features three functions, inspired by actual functions in our application corpus. The `find_document` function contains two separate database operations that filter documents by type, creation date, and optionally, the domain of the owner's email. The `create_document`, decorated with `@atomic`, executes as an interactive transaction. It creates a new document with the provided metadata and content on disk, and adds a corresponding entry to the `Document` model. The `confirm_email_change` function, also an interactive transaction, updates a user's profile with a pending email change after the user clicks a confirmation link containing the activation key.

5.2 Data collection

This section outlines the steps taken to gather the data for our study, which involves a collection of open-source applications. For each application, we annotate its individual database operations according to a set of rules described in later sections. Aggregated statistics from these annotations form the core of our dataset, presented in Appendix A.2. To facilitate this process, we developed a tool [2] that automates annotation for simple cases and provides a user interface for manual verification and handling of more complex scenarios that cannot be reliably automated.

5.2.1 Applications corpus

To compile the list of open-source applications, we leveraged the GitHub code search API and identified repositories containing keywords indicative of Django ORM usage

[2]The tool is written as an extension for Visual Studio Code and can be found at https://github.com/ctring/splinter

(e.g., `django.db`) or TypeORM usage (e.g., `createQueryBuilder`). Using GitHub stars as a measure of popularity, we narrowed out search to the most popular repositories, specifically those with at least 1000 stars for Django applications and at least 300 stars for TypeORM applications. We further refined the list by manually examining each repository and removing, for instance, those that only mention the keywords in their documentation without actually using any of the ORM frameworks, or those that are libraries themselves and ORM usage was only part of their example applications.

Our final corpus consists of 75 Django applications and 33 TypeORM applications[3]. These applications span a wide range of domains such as e-commerce, content management, customer relationship management, blogs, photo galleries, and more. The complete list of applications can be found in Appendix A.1. Although trends in closed-source applications might differ, we believe our corpus provides a representative sample among the open-source use cases.

Fig. 5.2 presents the number of models, operations, and interactive transactions across the application corpus. The applications are sorted in descending order of the number of operations (later graphs also present the applications in this order). We discovered 4,425 models and 30,091 transactions[4], including 1,487 interactive transactions (∼5% of the transactions). Each application has as low as a few models to a few hundreds of models. In general, an application has one to two orders of magnitude more operations than interactive transactions.

5.2.2 Semi-automated annotation

Manually annotating this large volume of items was impractical and prone to human error. On the other hand, automating some of our annotation rules required complex static analysis algorithms that might not consistently yield reliable results across

[3]The Django framework, released in 2005, predates TypeORM, which was released in 2016, thus there are more applications adopting Django than TypeORM.

[4]Recall that we define a transaction as either an interactive transaction or an operation that is not part of an interactive transaction in Section 5.1.

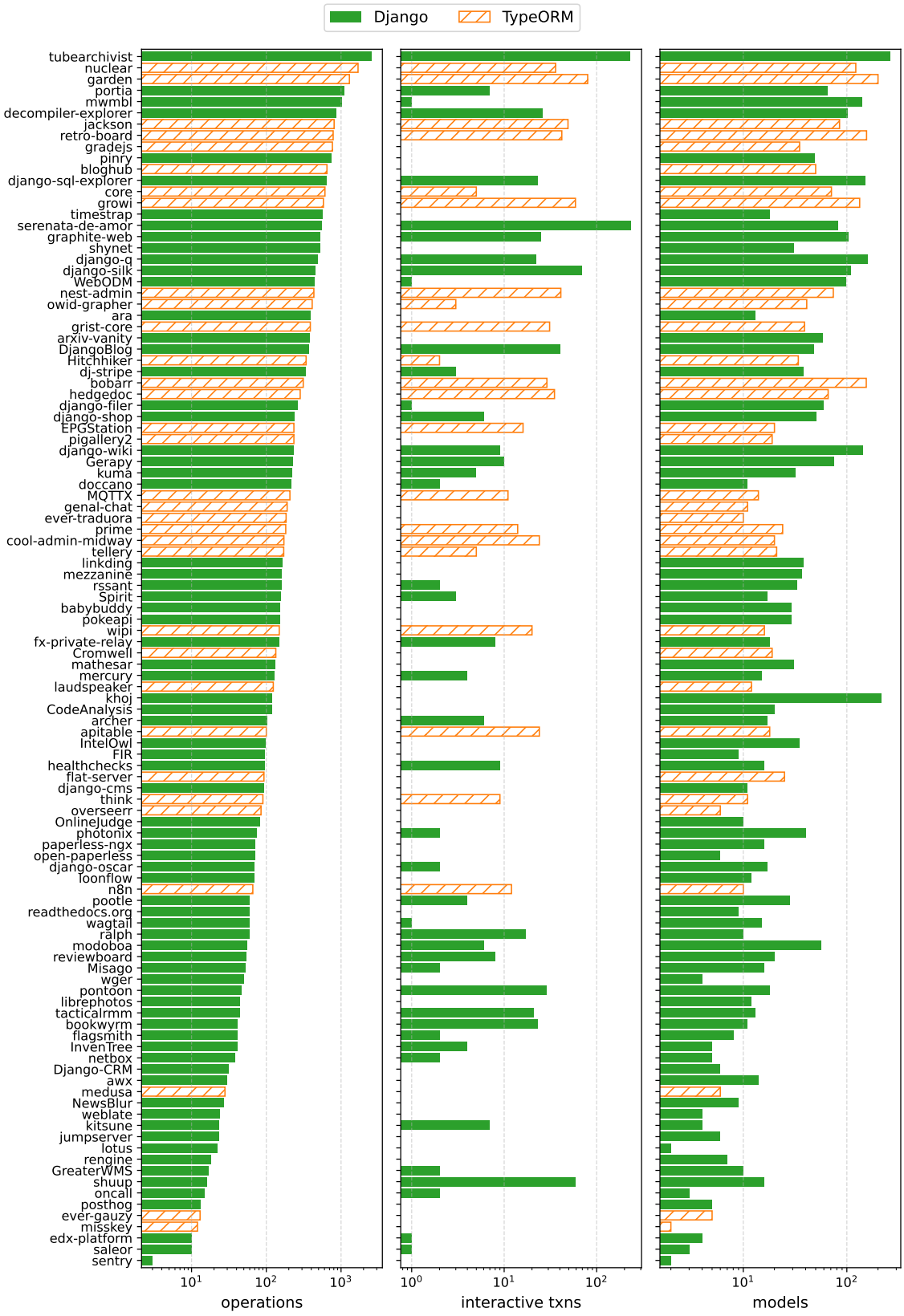


Figure 5.2: Number of models and transactions across applications.

diverse codebases with varying structures, dependencies, and coding conventions. Consequently, human verification remained necessary.

To address this challenge, we created a tool to facilitate both automated and manual annotation. This tool comprises a backend and a frontend.

The backend consists of a set of scripts tailored to each target programming language, which in our case are Python and TypeScript. Each script parses and traverses the abstract syntax trees (ASTs) of a given application. To accomplish this, we utilize the AST-related APIs provided by the *linters* of Python (mypy) and TypeScript (ESLint).

While traversing an AST, the scripts use keyword matching and type information to identify models and transactions. For example, the Python script identifies models by searching for classes that extend the `Model` base class and identifies transactions by matching method calls on objects of the `QuerySet` type to a fixed set of Django operations (e.g., `filter`, `get`, `update`). These ORM frameworks often use a lazy API, where a query is built by chaining multiple method calls and executed only upon certain *evaluating* methods. For instance, consider the following Django query

```
Document.objects.filter(type="paper")
                .filter(owner=1)
                .delete()
```

This query calls the `filter` method twice consecutively, but no query is sent to the database server until the `delete` method is called. The `delete` method triggers the construction of a query that deletes all tuples satisfying the conjunction of the two filtering predicates on attributes `type` and `owner`. Initially, our method matching script mistakenly identified these three method calls as separate operations. To resolve this, we reviewed all detected methods with the frontend and manually consolidated such cases into single operations. Method calls associated with the same operation are usually close to each other, making them easy to consolidate. However, in cases

where they are spread across multiple functions, we followed the chains of function calls to properly assemble them.

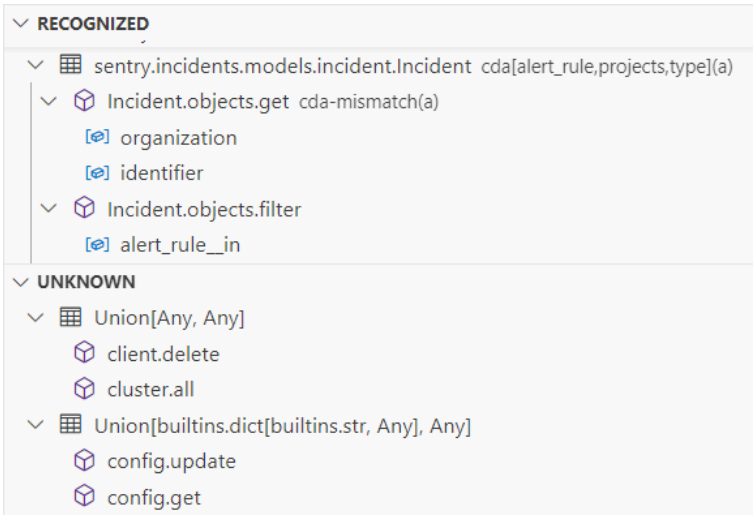


Figure 5.3: Screenshot of the data annotation tool. Each list shows the operations and their arguments. The operations are grouped by the types of the objects they are called on.

The frontend invokes the appropriate script for the opened codebase and displays the results in a graphical user interface, as shown in Fig. 5.3. Since Python and TypeScript lack strong typing, their type information may be incomplete, leading to the scripts not detecting all models and transactions. Therefore, the frontend presents two lists: `Recognized` for confidently identified items and `Unknown` for those matched based solely on keywords and to be reviewed by a human. Users can click on an item to highlight the corresponding code in the text editor, and move items between lists. Additionally, they can attach notes to the items for annotation purposes. The frontend also includes commands that automates all operation-level annotations mentioned in Section 5.3.1 (i.e. Complex Predicate, CDA Mismatch, and Non-equality Predicate), which covers the majority of the transactions. The remaining annotations still require manual intervention, but they are only applicable to interactive transactions, which are much fewer in number. For example, in Fig. 5.1, a human would need to examine the `create_document` and `confirm_email_change` functions to determine whether

they are Strictly or Non-strictly Interactive Transactions (Section 5.4.2), and whether they contain an Attribute Dependency (Section 5.3.1). The tool would handle the remaining annotations automatically.

5.3 Read/write set inferability

Some OLTP systems assume information about the read/write set of the transactions before executing them. They use the read/write set to analyze transaction conflicts in advance and in turn generate potentially more efficient execution schedules. In this section, we study the viability of efficiently extracting the read/write sets from real-world transactions prior to their execution.

5.3.1 Definition of a read/write set

There are multiple ways to define a read/write set, each with different implications for implementation complexity and concurrency. We focus on the definitions over the relational data model and SQL since they are used by the applications in this study.

The simplest method defines a read/write set by the relations a transaction accesses, which we refer to as the RELATION definition. SQL queries explicitly specify all accessed relations, making it straightforward to statically extract the read/write set for each transaction. However, this definition leads to conflicts at the relation level, even if the transactions access different sets of tuples, resulting in low concurrency.

At the other extreme[5], the read/write set can be defined by the predicates within a transaction and the relations they are evaluated on. We call this the PREDICATE definition. Transaction conflict can be determined similarly to *predicate locking* [44]. Specifically, the read/write set of a transaction is a list of triples (R, P, a) where R represents a relation, P is a predicate selecting tuples from R, and a indicates either

[5]Technically, this definition can be made more granular by including information about the projected attributes, but we do not consider such a definition since it is not done in practice.

a *read* or *write* access. The predicate P has the form *attr op expr*, where *attr* is an attribute of R, *expr* is an expression that can be evaluated to a scalar value without referring to any relation attribute, and *op* is one of $\{<, \leq, >, \geq, =, \neq\}$.

Two transactions T and T' are considered conflicting if there exist two triples (R, P, a) and (R', P', a') in the read/write set of T and T', respectively, satisfying all of the following:

1. $R = R'$
2. $a = write$ or $a' = write$
3. There exists a tuple t in R such that $P(t) \wedge P'(t) = true$

The PREDICATE definition allows for maximum information extraction from transaction code to detect conflicts, offering higher concurrency than the RELATION definition. Nevertheless, it is not always possible to infer the complete read/write set based solely on transaction code with the PREDICATE definition. In such cases (specified in a later paragraph), a system needs to employ *fallback mechanisms*.

One fallback approach is to use the RELATION definition for the problematic queries, effectively assuming they access the entire relation (i.e. $P = true$). Another technique is sending a reconnaissance query to reveal the exact tuples a transaction will access; the transaction may need to be aborted if the reconnoitered read/write set changes during execution. Regardless of the fallback mechanism used, they potentially reduce performance of the system and should be minimized.

With the PREDICATE definition, fallback mechanisms are necessary when a transaction contains a Complex Predicate or an Attribute Dependency.

Complex Predicates. Similar to previous works [44], [51], conflict detection for the PREDICATE definition restricts the predicates to a simple form to make deciding condition (3) tractable. In contrast, allowing general forms of the predicates makes condition (3) reducible from the Boolean satisfiability problem, which is NP-complete.

Therefore, a transaction requires fallback if it involves a complex predicate that does not follow P's restricted form. As an example, the second filter operation of the `find_document` function in Fig. 5.1 contains a complex predicate `endswith` that is true when the suffix of an owner's email matches the given domain.

Attribute Dependency. In a transaction, an attribute dependency arises when a query Q includes a predicate that compares an attribute with a value that can only be derived from the result of another query Q'. By this definition, Q's read/write set depends on values that must be obtained through the execution of the transaction (i.e. Q'). Therefore, fallback mechanisms are necessary when a transaction exhibits attribute dependencies. For instance, the `confirm_email_change` function has an attribute dependency, where a user is retrieved by their ID, which originates from a previously retrieved `PendingEmailChange` object.

On one hand, the RELATION definition limits transaction concurrency. On the other hand, usage of predicate-based concurrency control usage remains rare and primarily found in research prototypes due to its widely perceived high computational complexity [51]. Hence, neither of these definitions accurately represents existing upfront read/write set-based systems. Instead, these systems often define their read/write set based on primary keys (i.e. the predicate P only refers to primary key attributes) and rely on fallback mechanisms when a transaction selects tuples using non-primary key attributes. This approach generally results in higher concurrency than the RELATION definition and simpler implementations compared to the PREDICATE definition. To make this primary key-based definition relevant to a broader range of upfront read/write set-based systems, we introduce a new concept called **conflict detection attributes** (CDA) that subsumes primary keys, and define the read/write set based on CDA instead. We refer to this new definition as the CDA-BASED definition.

CDA of a relation is an ordered subset of the attributes of the relation and can be

defined on relations that do not have primary keys. Each relation has exactly one set of CDA. When using the CDA-Based definition, the read/write set of a transaction is represented by its predicates that access the the tuples only via the *prefixes* of their CDA. One approach to detect conflict with the CDA-Based definition is to create a multi-attribute index over the CDA of the relations. Queries that access based on prefixes of the CDA can acquire locks on entries in this index to detect conflict among them. For example, the CDA of the `Document` relation in Fig. 5.1 is the ordered subset `(type, create_date)`. If a transaction selects tuples in this relation with the predicate `type = "paper"`, it can lock all entries in the CDA index satisfying that predicate. Subsequent transactions selecting with `type = "paper"` and optionally an arbitrary value of `create_date` will be detected as conflict with the previous transaction.

The CDA-Based definition shares the same fallback scenarios as the Predicate definition, with one additional scenario:

CDA Mismatch. This occurs when a transaction accesses a relation without using a prefix of its CDA, in which cases, fallback mechanisms become necessary. In Fig. 5.1, the CDA of each model are represented by the shaded columns. The `create_document` and `confirm_email_change` functions contain user selection based on their email addresses rather than the attribute `id`, resulting in a CDA mismatch. Common sources of CDA Mismatches are transactions that perform lookups in non-CDA indexes (or secondary indexes when the CDA is the primary key).

The existence of non-equality operators (i.e. $<, >, \neq$) necessitates the use of more advanced data structures and optimizations to handle range queries efficiently. To investigate the feasibility of simplifying an upfront read/write set-based system by supporting only equality operators, we introduce another definition called CDAeq-Based. This definition is similar to the CDA-Based definition, but includes the following additional fallback scenario.

Non-equality Predicates. If the CDAEQ-BASED definition is used and there is an inequality predicate in the transaction, it needs to employ a fallback. For example, the `find_document` function selecting documents by a range of their creation dates is an example of non-equality predicates usage. While all Complex Predicates can be qualified as Non-equality Predicates, we do not count them in this category for ease of data interpretation.

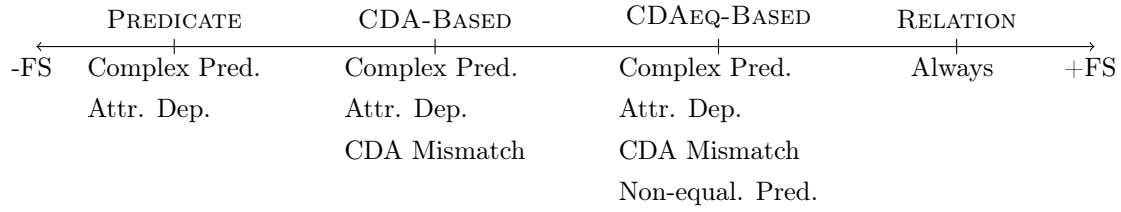


Figure 5.4: Read/write set definitions and their corresponding fallback scenarios (FS)

In conclusion, we present four different read/write set definitions, summarized in Fig. 5.4. The implementation complexity decreases from left to right on the chart, as does their potential efficiency due to the increasing number of fallback scenarios. However, it is important to note that the use of fallbacks, or the lack thereof, does not solely determine the overall performance of the system. Other factors, such as the overhead of evaluating the satisfiability of boolean expressions when using the PREDICATE definition, can also significantly influence performance. The key is to find a balance between the implementation overhead of a particular read/write definition and the frequency of its fallbacks. As the research community continues to develop new algorithms, they will evaluate the overheads of their own implementations, but they will make assumptions about the frequency of fallbacks. The purpose of our investigation is to help quantify these assumptions.

5.3.2 The need for fallback mechanisms

Our goal in this section is to estimate, for each read/write set definition, the proportion of transactions that require fallbacks in each application. To this end, we annotate all

transactions with the relevant fallback scenarios and aggregate their counts based on the specifications provided in Fig. 5.4.

For fallback scenarios other than CDA Mismatch, the information needed for annotation is self-contained within a transaction. However, CDA Mismatch requires assigning CDA to the relations. Therefore, we assigned each relation with the CDA that maximizes the number of transactions matching those CDA, then annotated the transactions accordingly. For instance, if the only operations on the `User` relation were those shown in Fig. 5.1, we would assigned (`email`) as the CDA for `User`, since it would result in two transactions matching the CDA and only one CDA Mismatch. However, we chose (`id`) instead for illustrative purposes[6]. Additionally, for queries involving joins, we perform the following transformation[7].

$$\sigma_{\beta}(R_1 \bowtie_{\theta} R_2) \equiv \sigma_{\beta_1 \wedge \beta_2 \wedge \ldots \wedge \beta_n}(R_1 \bowtie_{\theta} R_2) \tag{5.1}$$

$$\equiv \sigma_{\gamma_1 \wedge \gamma_2 \wedge \gamma_3}(R_1 \bowtie_{\theta} R_2) \tag{5.2}$$

$$\equiv \sigma_{\gamma_3}(\sigma_{\gamma_1}(R_1) \bowtie_{\theta} \sigma_{\gamma_2}(R_2)) \tag{5.3}$$

In equivalence (1), β is transformed into its conjunctive normal form $\beta_1 \wedge \beta_2 \wedge \ldots \wedge \beta_n$. In (2), these β's clauses are sorted into three groups γ_1, γ_2, and γ_3 such that γ_1 contains clauses that only refer to R_1, γ_2 contains clauses that only refer to R_2, and γ_3 contains clauses that refer to both R_1 and R_2. Finally, in (3), we use the selection distribution equivalence rule [64] to push the γ_1 and γ_2 predicates down to R_1 and R_2. The attributes of R_1 and R_2 that the query accessed are determined based on γ_1 and γ_2, respectively.

For instance, the second one-shot transaction in Fig. 5.1 is a join between the `Document` and the `User` relations. It does not exhibit CDA Mismatch with respect to

[6]This choice is still valid if we assume that `User` is more frequently accessed via `id` in other functions not displayed in the example.

[7]Projection is omitted as it is irrelevant

`Document` since it accesses the CDA prefix `(type, create_date)`, whereas it exhibits CDA Mismatch with respect to `User` since it accesses the attribute `email`, which is not a CDA prefix of `User`.

Fig. 5.5 illustrates the percentages of the fallback scenarios over the transactions in each application. Nearly all applications have CDA Mismatch ($105/108 \approx 97\%$) and Non-equality Predicates ($86/108 \approx 78\%$), with CDA Mismatch averaging 26.8% and Non-equality Predicates accounting for 3.6% of transactions per application. In contrast, Complex Predicate ($65/108 \approx 60\%$) and Attribute Dependency ($30/108 \approx 28\%$) are less common, averaging 1.7% and 0.7% of transactions per application, respectively. The bulk of Complex Predicates involve string pattern matching (e.g., `LIKE`), while the remaining few are predicates on non-scalar fields, such as checking whether an array contains a given value or whether a JSON object has a specific key.

Next, we use this annotation data to quantify the need for fallback mechanisms for the read/write set definitions. For each definition, we count the number of transactions containing one of the annotations from the corresponding column in Fig. 5.4. We then compute the percentage of such transactions for every application. The Relation definition is equivalent to performing fallbacks for all transactions; we include it here for completeness.

The fallback percentages for each definition are presented as cumulative distribution function (CDF) curves in Fig. 5.6. Each point on a curve corresponds to an application. From left to right, the curves follow the same order as in Fig. 5.4 since the percentage of transactions requiring fallbacks logically increases with the number of possible scenarios that trigger the fallbacks.

With the Predicate definition, one-third of applications do not require fallbacks for any of their transactions, and 90% of applications require fallbacks for less than 5.5% of their transactions. The CDA-Based definition's curve is shifted to the right significantly compared to the Predicate curve due to the addition of CDA Mismatch.

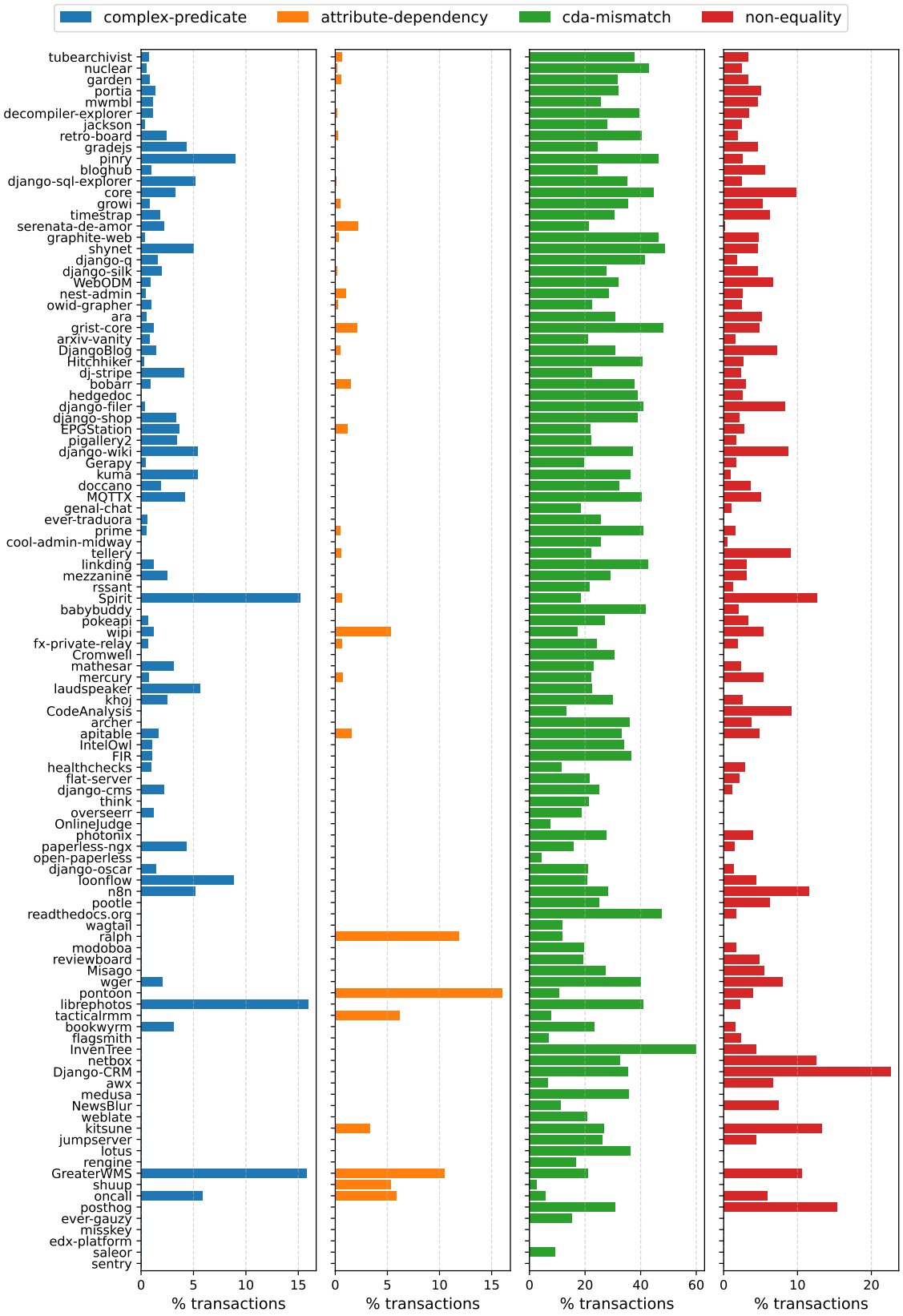


Figure 5.5: Percentages of the fallback scenarios.

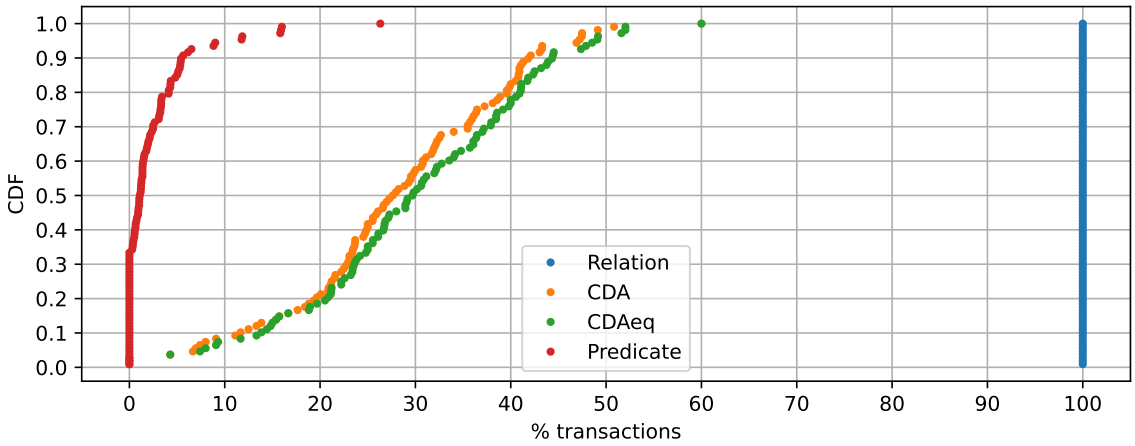


Figure 5.6: CDF of percentages of transactions requiring fallbacks for each read/write set definition.

As a result, applications are spread over a wider range of proportion of transactions requiring fallbacks, with only 3% of applications requiring no fallbacks, and 90% requiring fallbacks for less than 42% of their transactions. Finally, the CDAEQ-BASED definition's curve is shifted only slightly to the right of the CDA-BASED curve with the addition of Non-equality Predicates.

5.4 Transaction interactivity

Many modern OLTP system target applications that do not require the flexibility of interactive interfaces. These systems restrict applications to sending each transaction in a single round-trip, rather than allowing multiple round-trips within a single transaction. This section discusses the trade-offs of these approaches and presents results on their prevalence in real-world applications.

5.4.1 One-shot vs. interactive transactions

One-shot transactions are implemented either as precompiled stored procedures that are invoked from the client, or as code snippets sent to the database server for on-

the-fly interpretation. These transactions generally offer better performance for two reasons. First, if implemented as precompiled stored procedures, the SQL parsing and planning steps are done only once and can be skipped in subsequent invocations. Second, they reduce network I/O costs by requiring only one message round-trip, regardless of the number of database operations. Additionally, one-shot transactions enable new techniques that plan entire groups of transactions in advance, allowing for generation of more efficient execution schedules. For example, Janus [88], Aria [79], Caerus [56], and Detock (Chapter 4) reorder the transactions to avoid aborts, and Strife [99] clusters the transactions such that they can be executed in parallel without concurrency control.

However, one-shot transactions can add complexity to writing and maintaining application code. Instead of interleaving database access syntax with application logic, developers must write application logic and database access logic as separate code units. For database access logic, they must use a programming language supported by the database server, which may differ from the primary language used for the application. This requires developers to be familiar with additional database-specific languages and features. Moreover, this separation between application code and transaction code prevents transactions from performing tasks typically handled within the application process, such as accessing files on the application server or calling external APIs.

5.4.2 Annotations & results

In addition to determining whether a transaction is interactive, we aim to explore the feasibility of converting interactive transactions into one-shot transactions among the applications. We define an **external operation** as any operation that retrieves data from or induces a side effect outside the current process. Examples include file system actions such as creating or deleting files, making API calls to other services over the

network, or performing inter-process communication. We now introduce the following annotations for interactive transactions.

Strictly Interactive Transactions. We annotation a transaction as Strictly Interactive if it satisfies all of the following conditions:

1. It contains at least one external operation.
2. The invocation of the external operation depends on the result of a database operation.
3. The result of the external operation influences a database operation.

The `create_document` function in Fig. 5.1 serves as an example of a Strictly Interactive Transaction. It contains two external operations: `path.exists`, which checks for the existence of a file, and `write_file`, which creates a file at a specified path (condition (1)). The file path is constructed using the ID of a user retrieved from the database (condition (2)). The result of the file existence check determines whether a new document is inserted into the `Document` relation (condition (3)). Although it may seem there is no dependency between the `write_file` invocation and the insertion into the `Document` relation, if the `write_file` call fails (e.g., due to lack of write permission), the insertion will be rolled back. Therefore, a dependency exists between them.

The dependencies preclude the extraction of external operations from the transactions (see below for an example of such an extraction). On the other hand, while it is possible to extend the database system to support external operations via a plugin mechanism, it requires extra time and effort to develop and maintain such plugins. Furthermore, this approach shifts additional responsibilities to the database system, which could fundamentally change the overall organization of the entire application stack and introduce new security concerns. For example, if the `create_document` function were to be executed as a one-shot transaction within the database system,

either the file must reside on the same server running the database system, or the database server must access the file through a network file system or blob storage. From a software engineering perspective, this undermines the separation of concerns [135] between database systems, which manages structured data, and the components that handle blob data. Moreover, for a database system to access external APIs, it must be granted more permissions (e.g., via API keys or tokens) that it normally would not have, thereby increasing the attack surface and complicating security access management.

Non-strictly Interactive Transactions. We annotate a transaction as Non-strictly Interactive if it does not satisfy at least one of the above conditions, thus can be converted into one-shot transactions with reasonable effort.

The `confirm_email_change` function is an example where condition (1) is false since it does not contain any external operation.

If either (2) or (3) is not satisfied, it does not matter whether the external operations are performed inside or outside the transaction because the side effects of external operations are neither managed nor automatically rolled back by the database system. For instance, if the `path` variable in the `create_document` function depended on the user's email instead of the user ID, neither the `path.exists` nor the `write_file` call would depend on the `User.get` operation, failing condition (2). Consequently, the transaction would be annotated as Non-strictly Interactive as it could be rewritten with the file system accesses moved outside the transaction as follows.

```
def create_document(email, type, root_dir, name, content):
  path = root_dir / email / name
  if not path.exists():
    write_file(path, content)
    with atomic(): # the transaction starts here
      user = User.get(email=email)
      Document.create(type=type, create_date=today(),
                      path=path, owner=user.id)
```

With this rewrite, the file operations can be performed on the client side and the

transaction portion of the function contains only database operations.

Fig. 5.7 illustrates the distribution of one-shot, Strictly Interactive, and Non-strictly Interactive transactions across the applications. Out of the 108 applications, 65 applications (∼60%) contain interactive transactions (both Strictly and Non-strictly Interactive), and 19 applications (∼18%) have Strictly Interactive Transactions. Among the 65 applications with interactive transactions, an average of 9.4% of transactions per application are interactive. For the 19 applications with Strictly Interactive Transactions, an average of 2.6% of transactions per application are Strictly Interactive. The *retro-board* application is the only one where interactive transactions (∼79%) outnumber one-shot transactions (∼21%), yet all its interactive transactions can be converted into one-shot transactions.

We now dive deeper into the characteristics of Strictly Interactive Transactions. We observe that while an application may have multiple Strictly Interactive Transactions, each application is either dominated by or has only one type of external operation that gives rise to its Strictly Interactive Transactions. For example, if an application contains a Strictly Interactive Transaction involving calling an external API, it is almost always the case that other Strictly Interactive Transactions call that same API. Table 5.1 summarizes the percentage of Strictly Interactive Transactions and describes the dominating external operations across the 19 applications. Interestingly, the proportion of applications with Strictly Interactive Transactions is similar between Django ($13/75 \approx 17\%$) and TypeORM ($6/33 \approx 18\%$). Although Strictly Interactive Transactions are rare, the types of external operations involved are diverse, with no single type dominating the majority of applications.

Given that the applications in Table 5.1 are sorted in descending order of the number of their operations, we can see that Strictly Interactive Transactions tend to account for a larger proportion in applications with fewer operations (10 - 100). We hypothesize that this is because external operations are typically limited in scope.

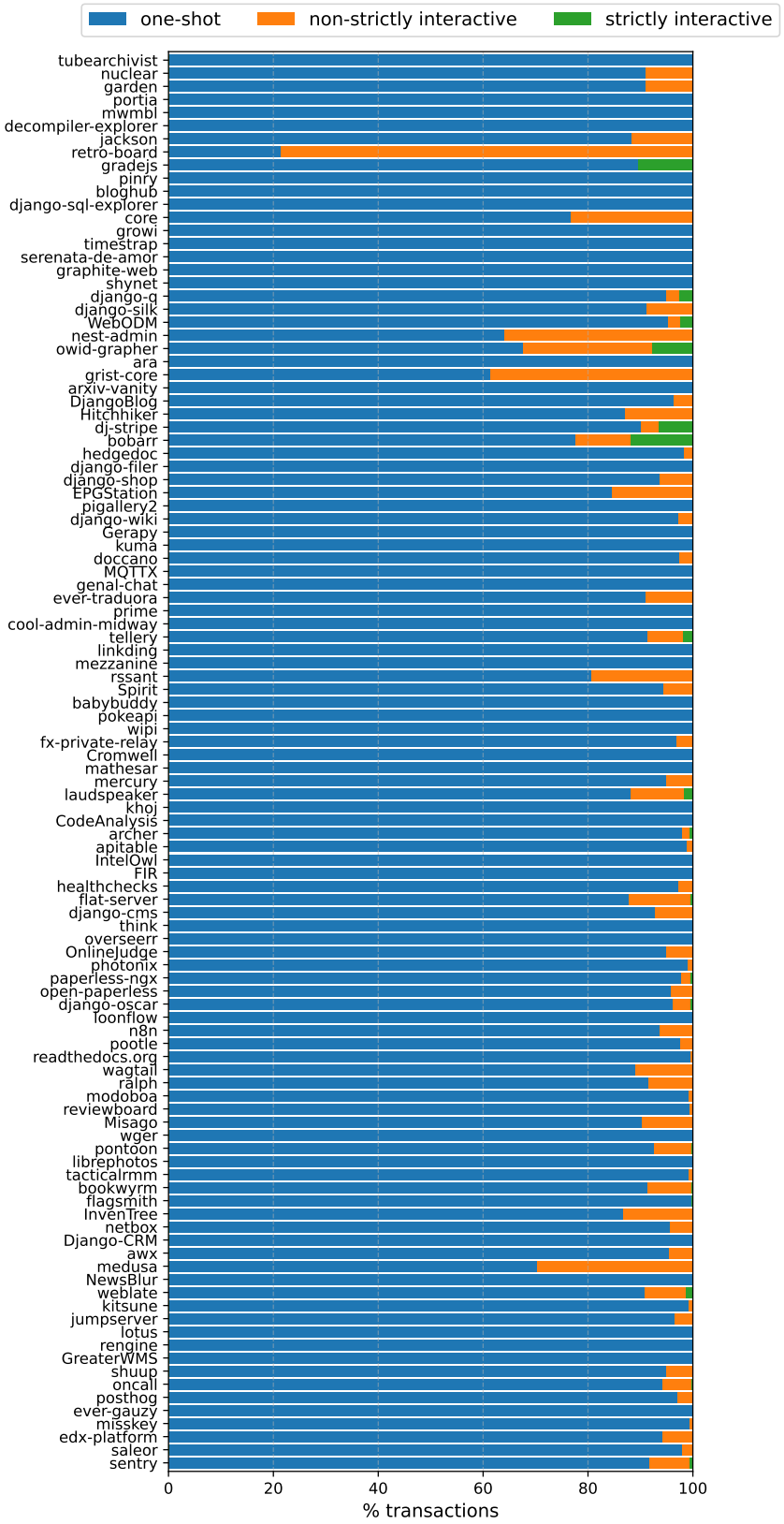


Figure 5.7: Transactions interactivity.

For instance, the number of API endpoints are usually fixed and minimized for easy management and to reduce the attack surface. External operations also need to be called only in a few specialized transactions (e.g., not all transactions send an email). Consequently, as an application scales, the number of transactions with external operations remains relatively constant, leading to a smaller proportion of these transactions in larger applications.

Table 5.1: Applications with strictly interactive transactions (SIT). TypeORM applications are marked with an asterisk (*).

Application	**% SIT**	**Description of external operations**
sentry	0.68	Making remote procedure calls
edx-platform	0.07	Sending emails
oncall	0.23	Sending data to message queues
weblate	1.41	Performing version control commit
flagsmith	0.23	Calling DynamoDB API
bookwyrm	0.21	Making HTTP requests
pontoon	0.24	Performing version control commit
django-oscar	0.42	Creating files
paperless-ngx	0.45	Creating/deleting files
flat-server*	0.51	Calling Redis API
archer	0.63	Sending emails
laudspeaker*	1.78	Sending data to message queues
tellery*	1.92	Sending emails
bobarr*	11.8	Creating/deleting files and directories
dj-stripe	6.56	Calling Stripe API
owid-grapher*	7.69	Appending data to files
WebODM	2.33	Copying files
django-q	2.5	Sending data to message queues
gradejs*	10.5	Calling Amazon S3 API

We identified 9 applications with interactive transactions that involve external operations but do *not* qualify as Strictly Interactive Transactions. Among these, 6 applications (*archer*, *flat-server*, *OnlineJudge*, *n8n*, *ralph*, and *kitsune*) contain transactions that do not meet condition (2), while 3 applications (*paperless-ngx*, *weblate*, and *pontoon*) contain transactions that do not satisfy condition (3).

Finally, we plot the percentage of interactive and strictly interactive transactions against the creation dates of the applications in Fig. 5.8. When combining both Django

and TypeORM applications, we observe an uptick in the use of interactive transactions starting around 2016, primarily driven by TypeORM applications. However, when analyzing Django and TypeORM applications separately, no distinct trend emerges within either group. Similarly, there is no discernible pattern in the adoption of strictly interactive transactions over time.

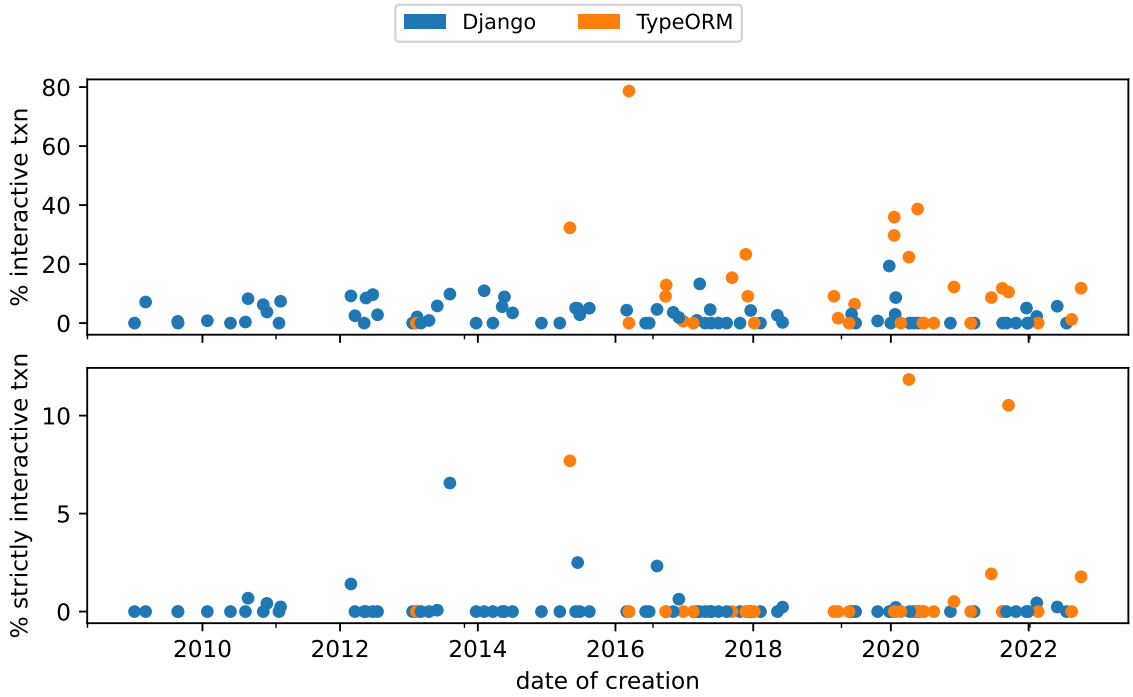


Figure 5.8: Transaction interactivity across application creation dates.

5.5 Discussions

Considering the new data presented in previous sections, we now discuss its implications for existing OLTP systems and suggest future research directions.

5.5.1 Read/write set inferability

The data in Section 5.3 paints a nuanced picture for systems that require upfront read/write sets. For 90% of applications, at least 58% of their transactions avoid the

need for fallback mechanism. While the portion of transactions requiring fallbacks is not insignificant, they are primarily due to CDA Mismatches. Unlike Attribute Dependency, which may involve analyzing chains of dependencies in multi-statement transactions and performing multiple OLLP lookups, inferring the read/write set for CDA Mismatches is simpler. It can be achieved by reading a non-CDA index to reveal the CDA of the selected tuples, resulting in a more lightweight fallback technique. Moreover, CDA Mismatches mostly occur in single-statement transactions, meaning the windows between optimistic reads and their validation are much shorter, reducing the likelihood of aborts in OLLP-type mechanisms.

It is clear that future research on improving OLLP or designing new fallback techniques should focus on minimizing CDA Mismatch frequency. A promising approach could involve moving towards a predicate-based definition of a read/write set, potentially implemented through predicate locking. Gaffney et. al [51] note that predicate locking is often overlooked by the research community due to its perceived high overhead. Yet, they demonstrate that it can be a viable concurrency control method with several optimizations that significantly enhance its performance and scalability. They further use predicate locking as a basis for modularizing transaction isolation into a separate service. Such modularization aligns well with the design of deterministic database systems [2], thus incorporating their techniques into deterministic database systems is an interesting research direction.

Lastly, the small gap between CDA-Based and CDAeq-Based suggests that it is reasonable for existing and future systems to initially support only equality predicates and use fallbacks when it comes to non-equality predicates. Nonetheless, most applications ($\sim$78%) contain non-equality predicates, which should eventually be addressed as the systems evolve.

5.5.2 Transaction interactivity

The data in Section 5.4 supports the notion that OLTP systems which do not support interactive transactions can still cater to a wide range of applications. Specifically, 40% of the applications in our corpus do not use interactive transactions at all. However, this also means that the remaining 60% of the applications, without further modifications, are unable to use any OLTP systems that support only one-shot transactions (referred to as *one-shot-only systems* in this section), many of which are listed in the beginning of this chapter. This presents opportunities for improvements in these systems.

One research direction is to embrace interactive transactions and extend the protocols of the one-shot-only systems to accommodate them. Recall that among applications involving interactive transactions, each application has, on average, 9.4% of its transactions being interactive. This suggests that interactive transactions likely constitute only a small portion of the live workload. Therefore, the extended protocols may not need to strive for the best performance for interactive transactions but should focus on *enabling* them while maintaining the original advantages of one-shot transactions. NCC [78] is an example for work that goes in this direction.

Another research direction is leveraging the fact that the vast majority of interactive transactions can be rewritten as one-shot transactions. Anecdotally, during our manual analysis, we rarely encountered transactions that were difficult to rewrite as stored procedures due to complexity or dependency on other libraries. However, previous surveys [57], [96] indicate that applications seldom use stored procedures because they introduce other challenges in the development process. Although significant research has focused on improving the performance of executing stored procedures, their usability has not advanced much, particularly in areas like versioning, debugging, and testing [53]. Hence, more research is needed to reduce the friction associated with employing them. One potential solution is to automate the synthesis of stored procedures from interactive transactions using static analysis, as proposed in WeBridge

[57]. For the rare cases of Strictly Interactive Transactions that cannot be easily converted into one-shot transactions, these transactions can remain unchanged, and, again, basic support for interactive transactions in one-shot-only systems should suffice as a fallback.

5.6 Conclusion

This empirical study investigated the prevalence of design assumptions underlying modern OLTP systems, specifically the trade-offs between transaction interactivity, read/write set inferability and efficiency. Our comprehensive analysis of open-source applications reveals that while read/write set inferability is achievable for a substantial number of transactions, fallback mechanisms are still necessary for many others, but can be designed to be lightweight. Furthermore, a significant portion of applications can operate effectively without interactive transactions, and many existing interactive transactions can be converted to one-shot transactions with minimal modifications. Lastly, we outlined future research directions based on the insights gained from this study.

Chapter 6

Related work

6.1 Scalable database system architectures

Shared-storage systems

SunStorm adopts a shared-storage architecture, similar to Aurora [129], [130], Socrates, [6], AlloyDB [89], PolarDB [22], [23], [140], [141], and Neon [90]. The advantages and disadvantages of this architecture have been discussed throughout Chapter 3.

Scaling write throughput in a shared-storage system can be done by supporting a multi-master setup, allowing different compute nodes to modify the same set of data items in the storage layer. Systems using this method exist both commercially (IBM Db2 pureScale, Oracle RAC, PolarDB-MP, and Amazon Aurora Limitless) and academically (Starry and Taurus MM).

IBM Db2 pureScale [126] and Oracle RAC [25], [93] transfer modified pages over the network instead of log records, causing high network utilization. PolarDB-MP [141] relies on a disaggregated shared memory (and storage) to provide all nodes with equal access to the data. They thus require high-bandwidth network or high-end hardware like RDMA [126], [141], which is not suitable for a multi-region deployment.

Amazon Aurora Limitless [146] adds a shard group consisting of data access nodes, each owning a portion of sharded tables or full copies of reference (non-sharded) tables. While this improves write throughput, writes from secondary regions still need to be forwarded to the primary region which incurs high latency.

Starry [150] allows each database node to send transactions to its local storage replica and uses a single sequencer replica to resolve conflicts via OCC. Taurus MM [36] uses vector-scalar clocks for the masters to see a consistent state and employs a hybrid page-row locking mechanism, which involves a global lock manager. Both systems rely on a centralized component to handle transactions, making them unsuitable for geo-partitioned deployments.

PolarDB-X [22] has a similar architecture as SunStorm. It deploys multiple single-primary systems across data centers (DC) and uses hybrid logical clock to provide snapshot isolation. However, it is optimized for running in DCs within close proximity (one millisecond round-trip time), and does not support serializable transactions.

To the best of our knowledge, SunStorm is the first to scale write throughput in an Aurora-style shared-storage architecture beyond a single geographic region.

Shared-nothing systems

The shared-nothing architecture, which has long been promoted as the go-to architecture for scaling OLTP workloads [115], has seen extensive research in geo-distributed transactions for both fully-replicated systems [31], [49], [67], [82], [88], [123], [138], [142], [149] and partially-replicated (geo-partitioned) systems [26], [41], [56], [91], [109], [118], [144]. The transaction execution methods of these systems fall into two camps: non-deterministic and deterministic. The systems in the first camp typically combine 2PC and a consensus protocol [71], [72], [92] to commit transactions at the end of their execution [26], [31], [41], [49], [67], [82], [118], [138], [142], [144], [149], while those in the second camp deterministically execute transactions based on an order

pre-established by a consensus protocol as described in Section 2.3 [56], [88], [91], [109], [123]. Both approaches are competitive in terms of latency. However, deterministic systems can handle more transactions per second especially under high contention workloads, but further research is needed to enhance their transaction expressivity. SunStorm falls in the non-deterministic camp as its use of traditional DBMSs aligns better with this approach.

6.2 Distributed transaction processing

Geo-replication

To achieve good latency and throughput, many geo-replicated systems use asynchronous replication and opt for weak consistency models such as eventual consistency [19], [35], [70], [73], [85], [97], [121], strong session serializability [34], timeline consistency [29], or causal consistency [37], [76], [77]. For stronger consistency (e.g. linearizability) over wide area network (WAN), the Paxos [71], [72] and Raft [92] consensus protocols are commonly implemented [31], [50], [82], [87], [109], [118], [123], [138]. These protocols require clients to send commands to a stable leader, causing a remote client to pay for extra WAN round-trip time. EPaxos [86] is a leaderless consensus protocol which involves tracking dependencies between commands and reordering strongly connected components, and has similarities to Detock's DDR algorithm. However, EPaxos has to rate limit to reduce the effect of livelock by prioritizing executing old commands over starting new commands. In contrast, Detock targets high conflict transactional workloads in which livelock would be common, and therefore uses opportunistic ordering instead.

Concurrency control

MDCC [67], Replicated Commit [82], TAPIR [149], Carousel [138], and Ocean Vista [49] are globally distributed database systems that aim to cut down the number of WAN round-trips but still incur cross-region latency for every transaction. In contrast, DETOCK processes transactions with strict serializability without incurring cross-region latency (except for multi-home transactions).

CockroachDB [118], [128], Spanner [31], Ocean Vista [49], and Dast [26] order transactions strictly by their timestamps. In contrast, DETOCK does not require global ordering, and only generates timestamps to reduce the probability of livelock, and can tolerate much large error bounds clock accuracy. Also, unlike Spanner, inaccurate clocks never affect the correctness of DETOCK.

RingBFT [105] uses a deadlock avoidance technique where it passes cross-shard transactions around the shards in a predetermined ring order, hence avoiding deadlocks and achieving high throughput. DETOCK instead allows distributed deadlocks to occur and resolves them on the fly, providing a new point in the tradeoff space when considering deadlock avoidance vs. detection for geo-partitioned systems. RingBFT can tolerate Byzantine failures, which comes with high latency, while DETOCK focuses on applications that benefit from low latency in non-Byzantine environments.

G-Store [33], L-Store [74], DynaMast [3], and MorphoSys [4] co-locate data in a single node using data migration or dynamic remastering to guarantee single-partition transactions. In contrast, home movement in DETOCK is a rare event, and is never required during transaction processing. Instead, it supports efficient multi-partition transactions.

Most proposed deterministic database systems cannot provide low-latency geo-distributed transactions due to reliance on a centralized global sequencing layer [62], [79], [103], [117], [123].

Previous work has proposed analyzing transaction dependencies to improve the

performance of concurrency control protocols [48], [87], [88], [134], [143]. Furthermore, dependency graphs have been used in practice for decades for deadlock detection [16]. Unlike traditional deadlock detection algorithms, DETOCK's DDR algorithm guarantees deterministic deadlock resolution, and therefore must generate a more complete dependency graph than traditional algorithms that only need to identify and destroy simple cycles. DDR minimizes the cost of this extra work by using opportunistic ordering to reduce the probability of deadlock.

6.3 Study on properties of transactions

Application studies

Faleiro's thesis [45] presents a similar study that examines the inferability of read/write sets across a smaller sample of 24 Ruby on Rails applications, without investigating their transaction interactivity. They primarily relied on automated static analysis and performed manual analysis on 6 applications. Their read/write set definition is a more restricted version of the CDA-BASED definition, with the CDA always being the primary key of each relation and only CDA Mismatch is considered for fallbacks. Interestingly, their CDF curve for flagged methods (analogous to transactions that requiring fallbacks in this work) closely resembles the CDA-BASED curve in Fig. 5.6. Specifically, 60% of their applications had fewer than 22% of methods flagged, and 80% had fewer than 37% flagged, compared to our findings where 60% of applications had fewer than 30% of transactions requiring fallbacks and 80% had fewer than 40% such transactions.

Other studies on application transactions in real-world applications have been done in the past. Bailis et. al [10] investigate concurrency control mechanisms that are implemented at application-level—or *feral* concurrency control—in 67 Ruby on Rails applications. Warszawski and Bailis [132] analyze traces of database activity and

identify security vulnerabilities stemming from weak isolation levels in 22 e-commerce applications. Tang et. al [119] study the characteristics, correctness, and performance of 91 *ad hoc* transactions among 8 web applications. Cheng et. al [28] examine 93 real-world concurrency bugs in database-backed applications to understand their root causes, consequences, and their fixes. Our work is orthogonal to these studies and provides comprehensive data on the interactivity and read/write set inferability aspects of transactions in real-world applications.

One-shot transactions

One approach to writing one-shot transactions is through specialized APIs. Sinfonia [5] introduces an API called *minitransactions*, each of which may contain operations such as comparing, reading, and writing specified data items. Similarly, Amazon DynamoDB [58] offers 3 transactional operations: `CheckItem`, `TransactGetItems`, and `TransactWriteItems`. Fauna [50] provides its own query language, FQL, inspired by TypeScript, that allows more expressive transactions with features like conditional branching (i.e., if-else). However, these APIs often come with limitations in expressiveness or require developers to learn new syntax. To avoid these issues, Thomson et. al [124] suggest a technique where developers write code snippets in a general-purpose programming language (e.g., Python) and submit them to the database system, which interprets the code on-the-fly.

Another approach to writing one-shot transactions is to use stored procedures, which are employed by nearly all systems discussed in the start of this chapter. However, Pavlo [96] found that most production database rarely use stored procedures, mainly because they pose challenges in software developments (e.g., DBAs are often reluctant to update them frequently). To address these challenges, WeBridge [57] proposes using static analysis to automate the generation of stored procedures from web application source code.

Read/write set inference

While conflict detection at relation level often results in a low degree of parallelism, IC3 [131] demonstrates that this information—easily obtained from the transaction code—can be leveraged to extract more concurrency in workloads whose transactions frequently access multiple relations at once. As mentioned in Section 5.3.1, systems relying on more fine-grained definitions of the read/write set, such as PREDICATE or CDA-BASED, may need to utilize OLLP [123]. Prognosticator [59] refines OLLP further by using symbolic execution to narrow down portion of the the read/write set that requires a reconnaissance query.

Chapter 7

Summary and discussion

7.1 Summary

The rise of cloud computing has enabled applications to reach users globally, but it has also introduced new challenges for database system practitioners and researchers. One of the biggest challenges is the high network delay caused by the physical distance between nodes in geographically distant regions. In a universe where nothing can exceed the speed of light, this delay is inevitable. Systems designed for such environments aim to minimize cross-region communications through geographic partitioning, placing data close to where it is most frequently accessed. The key differentiator for these systems, compared to simply deploying separate instances of a database system across different regions, is their support for *multi-region transactions*. This feature provides a seamless abstraction over the database, allowing users to access data from anywhere in the world through a single transactional interface. However, maintaining high performance and scalability while supporting multi-region transactions is a significant challenge. This thesis proposes two new solutions to address this problem and provides an application study to reassess the assumptions made by previous proposals.

Existing geo-partitioned database systems, such as CockroachDB and YugabyteDB,

incur cross-region communications not only at the commit phase of multi-region transactions but also during their execution, as remote reads need to fetch the latest data from other regions. Beside allowing scalability and low latency for Aurora-style systems, SunStorm also demonstrates that these remote reads can be performed locally on potentially stale copies of the data and validated at commit time. While this approach can result in aborts due to outdated reads, the reduction in cross-region communication substantially lowers the latency of multi-region transactions, as evidenced in Section 3.4.6. This reduction in latency decreases the conflict windows of these transactions, leading to lower abort rates compared to other systems.

Previous work on deterministic database systems, such as SLOG, has shown the advantages of determinism over geo-partitioned databases such as achieving high throughput under high-contention workloads. However, SLOG's multi-region transactions still suffer from high latency due to reliance on a centralized ordering process. Detock eliminates this ordering process entirely with its deterministic deadlock resolution protocol. With opportunistic ordering to avoid livelocks, Detock not only significantly reduces the latency of multi-region transactions but also maintains and even surpasses the throughput of SLOG, as demonstrated in Section 4.5.1.

A substantial body of research, including Detock, has focused on enhancing the performance and scalability of transaction processing by sacrificing transaction expressiveness, often by eliminating interactivity or requiring predefined read/write sets. Prior to the application study conducted in this thesis, these trade-offs had not been empirically validated. The study's data demonstrates that a significant number of transactions allow for easy read/write set inference. While many others still require fallback mechanisms, the implementation of the fallback mechanisms can be lightweight. Additionally, it supports the assumption that transaction interactivity is seldom used in real-world applications and most interactive transactions can be easily converted to one-shot transactions. Our study concludes with new directions

for future research in designing OLTP systems that balance transaction expressiveness and performance based on the collected data.

7.2 Discussion

To date, our work has focused on ensuring the highest levels of isolation and consistency—strict serializability [94]. This isolation level guarantees that transactions are executed in an order equivalent to their real-time submission, which provides the safest and most intuitive way to reason about concurrency in distributed systems. By enforcing this strong guarantee, developers can focus on application logic without worrying about concurrency anomalies. However, strict serializability inherently prevents the system from achieving high availability in the presence of network failures [9], [18]. Additionally, the rise of NoSQL systems in the early 2010s highlighted the demand for database systems that prioritize availability, delivering an "always-on" experience at the expense of weaker isolation and consistency guarantees. Researchers have explored various weaker yet practical isolation [11], [21], [27], [98], [114] and consistency levels [12], [73], [76], [97] to strike a balance between performance and usability. Nonetheless, there remains a gap in integrating these weaker guarantees into deterministic database systems—something that, to our knowledge, only one study has addressed [47]. This presents a promising opportunity for future research to explore incorporating weaker isolation and consistency levels into deterministic database systems.

While the research community has placed significant emphasis on enhancing transaction performance, much less attention has been paid to improving the usability of these systems. As discussed in Chapter 5, many new system proposals assume that applications rely on stored procedures for transaction execution. However, in practice, using stored procedures at scale introduces several challenges related to versioning, debugging, and testing [57], [96]. These hurdles often create friction for

developers, limiting the adoption of stored procedures despite their potential benefits. This represents a missed opportunity for performance gains, as simply encapsulating transactions within stored procedures can reduce communication overhead, which in some cases can account for up to 98% of the transaction runtime [65]. Therefore, there is a clear research opportunity in devising ways to make stored procedure usage more accessible and developer-friendly. On the other hand, interactive transactions offer greater flexibility in structuring applications. Supporting interactive transactions in systems that are optimized for one-shot transactions without sacrificing their performance advantages is another promising research direction.

Appendix A

Data from study on assumptions of modern transactional database systems

A.1 Application corpus

Each application is located at `https://github.com/<Owner>/<Repository>`.

Table A.1: Description of the applications in the corpus.

Owner/Repository	Description	Stars	Githash
getsentry/sentry	Monitoring	37712	669d6c3
saleor/saleor	E-commerce	20385	e78902b
openedx/edx-platform	Online courses	7120	765f2d8
misskey-dev/misskey	Blogging	9715	ae21b75
ever-co/ever-gauzy	Business management	2120	4ffd8ec
PostHog/posthog	Analytics	19823	c41995f
grafana/oncall	Incident management	3367	01cb87c
shuup/shuup	E-commerce	2214	25f78cf
GreaterWMS/GreaterWMS	Inventory management	3831	ec7db97
yogeshojha/rengine	Security	7245	9b6ccb3
uselotus/lotus	Pricing and billing engine	1699	0ea71f7
jumpserver/jumpserver	Access management	24734	4ebcba8
mozilla/kitsune	Support website	1276	2a939c5
WeblateOrg/weblate	Localization	4397	768f476
samuelclay/NewsBlur	Newsfeed	6787	56cd514
medusajs/medusa	E-commerce	24024	cff54d7

Table A.2: Description of the applications in the corpus (continued)

Owner/Repository	Description	Stars	Githash
ansible/awx	DevOps	13746	a15bcf1
MicroPyramid/Django-CRM	Customer relationship	1891	11184ce
netbox-community/netbox	Network automation	15492	0dde0b5
inventree/InvenTree	Inventory management	3960	6700a46
Flagsmith/flagsmith	Configuration management	4628	c896c50
bookwyrm-social/bookwyrm	Social network	2187	c4b21ee
amidaware/tacticalrmm	Monitoring	2981	8cfba49
LibrePhotos/librephotos	Photo management	6718	d0365b0
mozilla/pontoon	Localization	1437	fadf697
wger-project/wger	Fitness	2974	2b6c528
rafalp/Misago	Forum	2511	abad4f0
reviewboard/reviewboard	Code review	1547	d0e6a04
modoboa/modoboa	Mail hosting	2982	07b8cf9
allegro/ralph	Data center management	2197	5440253
wagtail/wagtail	Content management	17667	252bae9
readthedocs/readthedocs.org	Documentation	7950	757984f
translate/pootle	Localization	1487	1e3fc44
n8n-io/n8n	Workflow automation	42978	ee582cc
blackholll/loonflow	Workflow management	1904	b0e236b
django-oscar/django-oscar	E-commerce	6181	0247120
zhoubear/open-paperless	Document management	2559	b42d4e3
paperless-ngx/paperless-ngx	Document management	18332	3d56a56
photonixapp/photonix	Photo management	1822	8a02fb3
QingdaoU/OnlineJudge	Competitive programming	5960	c25a314
sct/overseerr	Media management	3604	77a33cb
fantasticit/think	Knowledge management	1934	1c2d133
django-cms/django-cms	Content management	10057	5ce5586
netless-io/flat-server	Classroom	638	4776c28
healthchecks/healthchecks	Monitoring	7687	46c70a6
certsocietegenerale/FIR	Incident management	1705	97fc077
intelowlproject/IntelOwl	Security	3203	f13a0d3
apitable/apitable	Project management	12731	404dedb
jly8866/archer	DevOps	1559	e208c19
Tencent/CodeAnalysis	Static code analysis	1626	386c948
khoj-ai/khoj	AI assistant	12103	ac3e508
laudspeaker/laudspeaker	Customer relationship	2056	9da9482
mljar/mercury	Notebook converter	3873	fc0d4b2
mathesar-foundation/mathesar	Database tool	2295	5bbcbbc
CromwellCMS/Cromwell	Content management	683	ce836a3
mozilla/fx-private-relay	Email address generator	1444	9d2ae6e
fantasticit/wipi	Blogging	615	9e3c15b
PokeAPI/pokeapi	Hobby	4067	f464535
babybuddy/babybuddy	Baby tracking	2020	ba46c49
nitely/Spirit	Forum	1160	069e82a
anyant/rssant	Newsfeed	1566	a6e37fb
stephenmcd/mezzanine	Content management	4747	e719286
sissbruecker/linkding	Bookmark management	5423	fa5f78c
tellery/tellery	Monitoring	352	0f0e1d2
cool-team-official/cool-admin-midway	Admin dashboard	2434	edc1383
birkir/prime	Content management	1718	336f50c
ever-co/ever-traduora	Translation management	1968	d1cd6bc

Table A.3: Description of the applications in the corpus (continued)

Owner/Repository	**Description**	**Stars**	**Githash**
genaller/genal-chat	Chat	1900	3c3c3bb
emqx/MQTTX	Connection management	3697	f589c82
doccano/doccano	Machine learning annotation	9272	4ac18ed
mdn/kuma	Documentation	1930	ae08600
Gerapy/Gerapy	Crawler management	3278	e5662e2
django-wiki/django-wiki	Knowledge management	1808	050f124
bpatrik/pigallery2	Photo management	1711	2aea6f4
l3tnun/EPGStation	Media management	538	c0e201b
awesto/django-shop	E-commerce	3187	13d9a77
django-cms/django-filer	File management	1732	0ae797c
hedgedoc/hedgedoc	Collaboration	4904	7286589
iam4x/bobarr	Media management	1476	0fdf960
dj-stripe/dj-stripe	Payment	1589	fa3990f
brookshi/Hitchhiker	Testing	2193	318bc47
liangliangyy/DjangoBlog	Blogging	6471	32158b3
arxiv-vanity/arxiv-vanity	Paper renderer	1599	f7eb2f1
gristlabs/grist-core	Online spreadsheet	6675	1152976
ansible-community/ara	DevOps	1827	8eda9c8
owid/owid-grapher	Data visualization	1356	c666571
wenqiyun/nest-admin	Permission management	577	ca53149
OpenDroneMap/WebODM	Image processing	2758	fa058e7
jazzband/django-silk	Profiling	4348	1cb4623
Koed00/django-q	Distributed task queue	1805	85baacc
milesmcc/shynet	Analytics	2870	c4410c4
graphite-project/graphite-web	Monitoring	5875	0ec7201
okfn-brasil/serenata-de-amor	Government data	4512	e7aeba7
overshard/timestrap	Time management	1683	a063302
weseek/growi	Collaboration	1297	941c546
ArkEcosystem/core	Blockchain	336	341c1e5
chrisclark/django-sql-explorer	Database tool	2695	a5d678b
shidenggui/bloghub	Blogging	413	314e309
pinry/pinry	Image board	3033	fd116a5
gradejs/gradejs	Security	407	ceca9c9
antoinejaussoin/retro-board	Collaboration	773	c22ada6
boxyhq/jackson	Authentication	1708	4177982
decompiler-explorer/decompiler-explorer	Decompiler	2000	0df1700
mwmbl/mwmbl	Search engine	1453	fd35442
scrapinghub/portia	Web scraping	9231	606467d
garden-io/garden	DevOps	3297	8ccd77f
nukeop/nuclear	Music streaming	11808	0da69f7
tubearchivist/tubearchivist	Media management	4476	0110736

A.2 Annotation data

Abbreviations used in this section

- M: Models
- O: Operations (outside interactive transactions)
- IT: Interactive Transactions
- SIT: Strictly Interactive Transactions
- CP: Complex Predicates
- AD: Attribute Dependency
- CM: CDA Mismatch
- NE: Non-equality
- TO: TypeORM application

Table A.4: Annotation data in descending order of operation count.

Owner/Repository	**M**	**O**	**IT**	**SIT**	**CP**	**AD**	**CM**	**NE**	**TO**
getsentry/sentry	260	2549	230	19	20	18	1046	93	
saleor/saleor	122	1688	36	0	8	3	740	43	
openedx/edx-platform	200	1294	80	1	11	8	435	46	
misskey-dev/misskey	65	1086	7	0	15	0	348	55	x
ever-co/ever-gauzy	141	1008	1	0	11	0	259	47	x
PostHog/posthog	101	852	26	0	10	2	346	30	
grafana/oncall	85	808	49	2	3	0	240	21	
shuup/shuup	155	784	42	0	20	2	333	16	
GreaterWMS/GreaterWMS	35	766	0	0	33	0	188	35	
yogeshojha/rengine	49	733	0	0	66	0	340	19	
uselotus/lotus	50	648	0	0	6	0	159	36	
jumpserver/jumpserver	150	640	23	0	34	1	233	16	
mozilla/kitsune	71	609	5	0	20	0	275	60	
WeblateOrg/weblate	133	581	59	9	5	3	227	34	
samuelclay/NewsBlur	18	565	0	0	10	0	173	35	
medusajs/medusa	82	551	233	0	17	17	167	1	x

Table A.5: Annotation data in descending order of operation count (continued).

Owner/Repository	M	O	IT	SIT	CP	AD	CM	NE	TO
ansible/awx	104	523	25	0	2	2	254	26	
MicroPyramid/Django-CRM	31	517	0	0	26	0	252	24	
netbox-community/netbox	160	480	22	0	8	0	208	9	
inventree/InvenTree	109	450	69	0	10	1	144	24	
Flagsmith/flagsmith	98	437	1	1	4	0	140	29	
bookwyrm-social/bookwyrm	74	433	41	1	2	5	135	12	
amidaware/tacticalrmm	41	411	3	0	4	1	93	10	
LibrePhotos/librephotos	13	392	0	0	2	0	121	20	
mozilla/pontoon	39	388	31	1	5	9	202	20	
wger-project/wger	59	378	0	0	3	0	79	6	
rafalp/Misago	48	374	40	0	6	2	127	30	
reviewboard/reviewboard	34	341	2	0	1	0	139	9	
modoboa/modoboa	38	337	3	0	14	0	76	8	
allegro/ralph	154	312	29	0	3	5	129	10	
wagtail/wagtail	66	284	35	0	0	0	124	8	
readthedocs/readthedocs.org	60	263	1	0	1	0	108	22	
translate/pootle	51	235	6	0	8	0	94	5	
n8n-io/n8n	20	234	16	0	9	3	55	7	x
blackholll/loonflow	19	234	0	0	8	0	52	4	
django-oscar/django-oscar	144	231	9	1	13	0	89	21	
zhoubear/open-paperless	75	224	10	0	1	0	46	4	
paperless-ngx/paperless-ngx	32	218	5	1	12	0	81	2	
photonixapp/photonix	11	215	2	0	4	0	70	8	
QingdaoU/OnlineJudge	14	207	11	0	9	0	88	11	
sct/overseerr	11	190	0	0	0	0	35	2	x
fantasticit/think	10	184	0	0	1	0	47	0	x
django-cms/django-cms	24	182	14	0	1	1	80	3	
netless-io/flat-server	20	172	24	1	0	0	50	1	x
healthchecks/healthchecks	21	171	5	0	0	1	39	16	
certsocietegenerale/FIR	38	164	0	0	2	0	70	5	
intelowlproject/IntelOwl	37	161	0	0	4	0	47	5	
apitable/apitable	33	160	2	0	0	0	35	2	x
jly8866/archer	17	155	3	1	24	1	29	20	
Tencent/CodeAnalysis	29	153	0	0	0	0	64	3	
khoj-ai/khoj	29	152	0	0	1	0	41	5	
laudspeaker/laudspeaker	16	149	20	3	2	9	29	9	x
mljar/mercury	18	149	8	0	1	1	38	3	
mathesar-foundation/mathesar	19	134	0	0	0	0	41	0	
CromwellCMS/Cromwell	31	130	0	0	4	0	30	3	x
mozilla/fx-private-relay	15	127	4	0	1	1	29	7	
fantasticit/wipi	12	124	0	0	7	0	28	0	x
PokeAPI/pokeapi	217	120	0	0	3	0	36	3	
babybuddy/babybuddy	20	120	0	0	0	0	16	11	
nitely/Spirit	17	102	6	0	0	0	39	4	
anyant/rssant	18	100	24	0	2	2	41	6	
stephenmcd/mezzanine	35	97	0	0	1	0	33	0	
sissbruecker/linkding	9	96	0	0	1	0	35	0	
tellery/tellery	16	95	9	2	1	0	12	3	x
cool-team-official/cool-admin-midway	25	93	0	0	0	0	20	2	x
birkir/prime	11	92	0	0	2	0	23	1	x
ever-co/ever-traduora	11	90	9	0	0	0	21	0	x

Table A.6: Annotation data in descending order of operation count (continued).

Owner/Repository	M	O	IT	SIT	CP	AD	CM	NE	TO
genaller/genal-chat	6	85	0	0	1	0	16	0	x
emqx/MQTTX	10	81	0	0	0	0	6	0	x
doccano/doccano	40	74	2	0	0	0	21	3	
mdn/kuma	16	70	0	0	3	0	11	1	
Gerapy/Gerapy	6	70	0	0	0	0	3	0	
django-wiki/django-wiki	17	69	2	0	1	0	15	1	
bpatrik/pigallery2	12	68	0	0	6	0	14	3	x
l3tnun/EPGStation	10	66	12	0	4	0	22	9	x
awesto/django-shop	28	60	4	0	0	0	16	4	
django-cms/django-filer	9	59	0	0	0	0	28	1	
hedgedoc/hedgedoc	15	59	1	0	0	0	7	0	x
iam4x/bobarr	10	59	17	9	0	9	9	0	x
dj-stripe/dj-stripe	56	55	6	4	0	0	12	1	
brookshi/Hitchhiker	20	54	8	0	0	0	12	3	x
liangliangyy/DjangoBlog	16	53	2	0	0	0	15	3	
arxiv-vanity/arxiv-vanity	4	50	0	0	1	0	20	4	
gristlabs/grist-core	18	46	29	0	0	12	8	3	x
ansible-community/ara	12	44	0	0	7	0	18	1	
owid/owid-grapher	13	44	21	5	0	4	5	0	x
wenqiyun/nest-admin	11	41	23	0	2	0	15	1	x
OpenDroneMap/WebODM	8	41	2	1	0	0	3	1	
jazzband/django-silk	5	41	4	0	0	0	27	2	
Koed00/django-q	5	38	2	1	0	0	13	5	
milesmcc/shynet	6	31	0	0	0	0	11	7	
graphite-project/graphite-web	14	30	0	0	0	0	2	2	
okfn-brasil/serenata-de-amor	6	28	0	0	0	0	10	0	
overshard/timestrap	9	27	0	0	0	0	3	2	
weseek/growi	4	24	0	0	0	0	5	0	x
ArkEcosystem/core	4	23	7	0	0	1	8	4	x
chrisclark/django-sql-explorer	6	23	0	0	0	0	6	1	
shidenggui/bloghub	2	22	0	0	0	0	8	0	x
pinry/pinry	7	18	0	0	0	0	3	0	
gradejs/gradejs	10	17	2	2	3	2	4	2	x
antoinejaussoin/retro-board	16	16	59	0	0	4	2	0	x
boxyhq/jackson	3	15	2	0	1	1	1	1	x
decompiler-explorer/decompiler-explorer	5	13	0	0	0	0	4	2	
mwmbl/mwmbl	5	13	0	0	0	0	2	0	
scrapinghub/portia	2	12	0	0	0	0	0	0	
garden-io/garden	4	10	1	0	0	0	0	0	x
nukeop/nuclear	3	10	1	0	0	0	1	0	x
tubearchivist/tubearchivist	2	3	0	0	0	0	0	0	

Bibliography

[1] D. Abadi, "Consistency tradeoffs in modern distributed database system design: CAP is only part of the story," *Computer*, vol. 45, no. 2, pp. 37–42, Feb. 2012, ISSN: 0018-9162, 1558-0814. DOI: 10.1109/MC.2012.33.

[2] D. J. Abadi and J. M. Faleiro, "An overview of deterministic database systems," en, *Commun. ACM*, vol. 61, no. 9, pp. 78–88, Aug. 2018, ISSN: 0001-0782, 1557-7317. DOI: 10.1145/3181853.

[3] M. Abebe, B. Glasbergen, and K. Daudjee, "DynaMast: Adaptive dynamic mastering for replicated systems," in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, vol. 00, Dallas, TX, USA: IEEE, Apr. 2020, pp. 1381–1392, ISBN: 9781728129037. DOI: 10.1109/ICDE48307.2020.00123.

[4] M. Abebe, B. Glasbergen, and K. Daudjee, "MorphoSys: Automatic physical design metamorphosis for distributed database systems," en, *Proceedings VLDB Endowment*, vol. 13, no. 13, pp. 3573–3587, Sep. 2020, ISSN: 2150-8097, 2150-8097. DOI: 10.14778/3424573.3424578.

[5] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis, "Sinfonia: A new paradigm for building scalable distributed systems," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, ser. SOSP '07, vol. 41, Stevenson, Washington, USA: Association for Computing Machinery, Oct. 2007, pp. 159–174, ISBN: 9781595935915. DOI: 10.1145/1294261.1294278.

[6] P. Antonopoulos, A. Budovski, C. Diaconu, *et al.*, "Socrates: The new SQL server in the cloud," in *Proceedings of the 2019 International Conference on Management of Data*, ser. SIGMOD '19, Amsterdam, Netherlands: Association for Computing Machinery, Jun. 2019, pp. 1743–1756, ISBN: 9781450356435. DOI: 10.1145/3299869.3314047.

[7] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, *et al.*, "System r: Relational approach to database management," en, *ACM Trans. Database Syst.*, vol. 1, no. 2, pp. 97–137, Jun. 1976, ISSN: 0362-5915, 1557-4644. DOI: 10.1145/320455.320457.

[8] "Aws - amazon aurora multi-master is now generally available." (2024), [Online]. Available: https://aws.amazon.com/about-aws/whats-new/2019/08/amazon-aurora-multimaster-now-generally-available/.

[9] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Highly available transactions: Virtues and limitations," en, *Proceedings VLDB Endowment*, vol. 7, no. 3, pp. 181–192, Nov. 2013, ISSN: 2150-8097. DOI: 10.14778/2732232.2732237.

[10] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Feral concurrency control: An empirical investigation of modern application integrity," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15, Melbourne, Victoria, Australia: Association for Computing Machinery, May 2015, pp. 1327–1342, ISBN: 9781450327589. DOI: 10.1145/2723372.2737784.

[11] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Scalable atomic visibility with RAMP transactions," *ACM Trans. Database Syst.*, vol. 41, no. 3, pp. 1–45, Jul. 2016, ISSN: 0362-5915. DOI: 10.1145/2909870.

[12] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Bolt-on causal consistency," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, New York New York USA: ACM, Jun. 2013, pp. 761–772, ISBN: 9781450320375. DOI: 10.1145/2463676.2465279.

[13] J. Baker, C. Bond, J. C. Corbett, *et al.*, "Megastore: Providing scalable, highly available storage for interactive services," in *Proceedings of the Conference on Innovative Data system Research (CIDR)*, 2011, pp. 223–234.

[14] P. A. Bernstein, D. W. Shipman, and W. S. Wong, "Formal aspects of serializability in database concurrency control," *IEEE Trans. Software Eng.*, vol. SE-5, no. 3, pp. 203–216, May 1979, ISSN: 0098-5589, 1939-3520. DOI: 10.1109/tse.1979.234182.

[15] P. A. Bernstein and N. Goodman, "Concurrency control in distributed database systems," *ACM Computing Surveys (CSUR)*, vol. 13, no. 2, pp. 185–221, 1981, ISSN: 0360-0300. DOI: 10.1145/356842.356846.

[16] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, en. Addison-Wesley Publishing Company, 1987, ISBN: 9780201107159.

[17] Y. Breitbart, H. Garcia-Molina, and A. Silberschatz, "Overview of multidatabase transaction management," *VLDB J.*, vol. 1, no. 2, pp. 181–239, Oct. 1992, ISSN: 1066-8888, 0949-877X. DOI: 10.1007/BF01231700.

[18] E. Brewer, "Towards robust distributed systems," *PODC*, p. 7, Jul. 2000. DOI: 10.1145/343477.343502.

[19] N. Bronson, Z. Amsden, G. Cabrera, *et al.*, "TAO: Facebook's distributed data store for the social graph," in *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'13, San Jose, CA: USENIX Association, 2013, pp. 49–60.

[20] J. Brutlag, "Speed matters for google web search," *Google. June*, vol. 2, no. 9, 2009.

[21] M. J. Cahill, U. Röhm, and A. D. Fekete, "Serializable isolation for snapshot databases," en, *ACM Trans. Database Syst.*, vol. 34, no. 4, pp. 1–42, Dec. 2009, ISSN: 0362-5915, 1557-4644. DOI: 10.1145/1620585.1620587.

[22] W. Cao, F. Li, G. Huang, *et al.*, "PolarDB-X: An elastic distributed relational database for Cloud-Native applications," in *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, IEEE, May 2022, pp. 2859–2872, ISBN: 9781665408837, 9781665408844. DOI: 10.1109/ICDE53745.2022.00259.

[23] W. Cao, Y. Zhang, X. Yang, *et al.*, "PolarDB serverless: A cloud native database for disaggregated data centers," in *Proceedings of the 2021 International Conference on Management of Data*, ser. SIGMOD '21, Virtual Event, China: Association for Computing Machinery, Jun. 2021, pp. 2477–2489, ISBN: 9781450383431. DOI: 10.1145/3448016.3457560.

[24] P. Chairunnanda, K. Daudjee, and M. T. Özsu, "ConfluxDB: Multi-master replication for partitioned snapshot isolation databases," en, *Proceedings VLDB Endowment*, vol. 7, no. 11, pp. 947–958, Jul. 2014, ISSN: 2150-8097. DOI: 10.14778/2732967.2732970.

[25] Chandrasekaran and Bamford, "Shared cache - the future of parallel databases," in *Proceedings 19th International Conference on Data Engineering*, vol. 0, Mar. 2003, p. 840. DOI: 10.1109/ICDE.2003.1260883.

[26] X. Chen, H. Song, J. Jiang, *et al.*, "Achieving low tail-latency and high scalability for serializable transactions in edge computing," in *Proceedings of the Sixteenth European Conference on Computer Systems*, ser. EuroSys '21, Online Event, United Kingdom: Association for Computing Machinery, Apr. 2021, pp. 210–227, ISBN: 9781450383349. DOI: 10.1145/3447786.3456238.

[27] A. Cheng, X. Shi, L. Pan, *et al.*, "RAMP-TAO: Layering atomic transactions on facebook's online TAO data store," en, *Proceedings VLDB Endowment*, vol. 14, no. 12, pp. 3014–3027, Jul. 2021, ISSN: 2150-8097. DOI: 10.14778/3476311.3476379.

[28] C.-W. Cheng, M. Han, N. Xu, S. Blanas, M. D. Bond, and Y. Wang, "Developer's responsibility or database's responsibility? rethinking concurrency control in databases," *CIDR*, 2023.

[29] B. F. Cooper, R. Ramakrishnan, U. Srivastava, *et al.*, "PNUTS: Yahoo!'s hosted data serving platform," *Proceedings VLDB Endowment*, vol. 1, no. 2, pp. 1277–1288, Aug. 2008, ISSN: 2150-8097. DOI: 10.14778/1454159.1454167.

[30] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM symposium on Cloud computing*, Indianapolis Indiana USA: ACM, Jun. 2010, pp. 143–154, ISBN: 9781450300360. DOI: 10.1145/1807128.1807152.

[31] J. C. Corbett, J. Dean, M. Epstein, *et al.*, "Spanner: Google's globally distributed database," en, *ACM Trans. Comput. Syst.*, vol. 31, no. 3, pp. 1–22, Aug. 2013, ISSN: 0734-2071, 1557-7333. DOI: 10.1145/2491245.

[32] J. Cowling and B. Liskov, "Granola:{Low-Overhead} distributed transaction coordination," in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 2012, pp. 223–235.

[33] S. Das, D. Agrawal, and A. El Abbadi, "G-Store: A scalable data store for transactional multi key access in the cloud," in *Proceedings of the 1st ACM symposium on Cloud computing*, ser. SoCC '10, Indianapolis, Indiana, USA: Association for Computing Machinery, Jun. 2010, pp. 163–174, ISBN: 9781450300360. DOI: 10.1145/1807128.1807157.

[34] K. Daudjee and K. Salem, "Lazy database replication with ordering guarantees," 2004, pp. 424–435. DOI: 10.1109/ICDE.2004.1320016.

[35] G. DeCandia, D. Hastorun, M. Jampani, *et al.*, "Dynamo: Amazon's highly available key-value store," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, Stevenson Washington USA: ACM, Oct. 2007, p. 205, ISBN: 9781595935915. DOI: 10.1145/1294261.1294281.

[36] A. Depoutovitch, C. Chen, P.-A. Larson, *et al.*, "Taurus MM: Bringing Multi-Master to the cloud," en, *Proceedings VLDB Endowment*, vol. 16, no. 12, pp. 3488–3500, Aug. 2023, ISSN: 2150-8097. DOI: 10.14778/3611540.3611542.

[37] D. Didona, R. Guerraoui, J. Wang, and W. Zwaenepoel, "Causal consistency and latency optimality: Friend or foe?" *Proceedings VLDB Endowment*, vol. 11, no. 11, pp. 1618–1632, Jul. 2018, ISSN: 2150-8097. DOI: 10.14778/3236187.3236210.

[38] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux, "OLTP-Bench: An extensible testbed for benchmarking relational databases," *Proceedings VLDB Endowment*, vol. 7, no. 4, pp. 277–288, Dec. 2013, ISSN: 2150-8097. DOI: 10.14778/2732240.2732246.

[39] Z. Dong, Z. Wang, X. Zhang, *et al.*, "Fine-Grained Re-Execution for efficient batched commit of distributed transactions," *Proceedings VLDB Endowment*, vol. 16, no. 8, pp. 1930–1943, Apr. 2023, ISSN: 2150-8097. DOI: 10.14778/3594512.3594523.

[40] J. Du, S. Elnikety, and W. Zwaenepoel, "Clock-SI: Snapshot isolation for partitioned data stores using loosely synchronized clocks," in *2013 IEEE 32nd International Symposium on Reliable Distributed Systems*, Braga, Portugal: IEEE, Sep. 2013, pp. 173–184, ISBN: 9780769551159. DOI: 10.1109/srds.2013.26.

[41] T. Eldeeb, P. A. Bernstein, A. Cidon, and J. Yang, "Chablis: Fast and general transactions in Geo-Distributed systems," in *CIDR*, 2024.

[42] T. Eldeeb, X. Xie, P. A. Bernstein, A. Cidon, and J. Yang, "Chardonnay: Fast and general datacenter transactions for On-Disk databases," in *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, 2023, pp. 343–360.

[43] V. Enes, C. Baquero, A. Gotsman, and P. Sutra, "Efficient replication via timestamp stability," in *Proceedings of the Sixteenth European Conference on Computer Systems*, ser. EuroSys '21, Online Event, United Kingdom: Association for Computing Machinery, Apr. 2021, pp. 178–193, ISBN: 9781450383349. DOI: 10.1145/3447786.3456236.

[44] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The notions of consistency and predicate locks in a database system," *Commun. ACM*, vol. 19, no. 11, pp. 624–633, 1976, ISSN: 0001-0782. DOI: 10.1145/360363.360369.

[45] J. Faleiro, "High performance multi-core transaction processing via deterministic execution," Ph.D. dissertation, 2018.

[46] J. M. Faleiro and D. J. Abadi, "Rethinking serializable multiversion concurrency control," en, *Proceedings VLDB Endowment*, vol. 8, no. 11, pp. 1190–1201, Jul. 2015, ISSN: 2150-8097, 2150-8097. DOI: 10.14778/2809974.2809981.

[47] J. M. Faleiro, D. J. Abadi, and J. M. Hellerstein, "High performance transactions via early write visibility," en, *Proceedings VLDB Endowment*, vol. 10, no. 5, pp. 613–624, Jan. 2017, ISSN: 2150-8097, 2150-8097. DOI: 10.14778/3055540.3055553.

[48] J. M. Faleiro, A. Thomson, and D. J. Abadi, "Lazy evaluation of transactions in database systems," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, Snowbird Utah USA: ACM, Jun. 2014, pp. 15–26, ISBN: 9781450323765. DOI: 10.1145/2588555.2610529.

[49] H. Fan and W. Golab, "Ocean vista: Gossip-based visibility control for speedy geo-distributed transactions," en, *Proceedings VLDB Endowment*, vol. 12, no. 11, pp. 1471–1484, Jul. 2019, ISSN: 2150-8097. DOI: 10.14778/3342263.3342627.

[50] *Fauna*, en, https://fauna.com/, Accessed: 2024-7-28.

[51] K. P. Gaffney, R. Claus, and J. M. Patel, "Database isolation by scheduling," en, *Proceedings VLDB Endowment*, vol. 14, no. 9, pp. 1467–1480, May 2021, ISSN: 2150-8097. DOI: 10.14778/3461535.3461537.

[52] R. Guerraoui and A. Schiper, "The decentralized Non-Blocking atomic commitment protocol," in *Proceedings.Seventh IEEE Symposium on Parallel and Distributed Processing*, IEEE, 1995, pp. 2–9, ISBN: 9780818671951. DOI: 10.1109/SPDP.1995.530658.

[53] S. Gupta and K. Ramachandra, "Procedural extensions of SQL: Understanding their usage in the wild," *Proceedings VLDB Endowment*, vol. 14, no. 8, pp. 1378–1391, Apr. 2021, ISSN: 2150-8097. DOI: 10.14778/3457390.3457402.

[54] R. Harding, D. Van Aken, A. Pavlo, and M. Stonebraker, "An evaluation of distributed concurrency control," en, *Proceedings VLDB Endowment*, vol. 10, no. 5, pp. 553–564, Jan. 2017, ISSN: 2150-8097. DOI: 10.14778/3055540.3055548.

[55] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," en, *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, Jul. 1990, ISSN: 0164-0925, 1558-4593. DOI: 10.1145/78969.78972.

[56] J. Hildred, M. Abebe, and K. Daudjee, "Caerus: Low-Latency distributed transactions for Geo-Replicated systems," *Proceedings VLDB Endowment*, vol. 17, no. 3, pp. 469–482, Jan. 2024, ISSN: 2150-8097. DOI: 10.14778/3632093.3632109.

[57] G. Hu, Z. Wang, C. Tang, *et al.*, "WeBridge: Synthesizing stored procedures for Large-Scale Real-World web applications," *Proc. ACM SIGMOD Int. Conf. Manag. Data*, vol. 2, no. 1, pp. 1–29, Mar. 2024, ISSN: 0730-8078. DOI: 10.1145/3639319.

[58] J. Idziorek, A. Keyes, C. Lazier, *et al.*, "Distributed transactions at scale in amazon DynamoDB," *Proc. USENIX Annu. Tech. Conf.*, pp. 705–717, 2023.

[59] S. Issa, M. Viegas, P. Raminhas, N. Machado, M. Matos, and P. Romano, "Exploiting symbolic execution to accelerate deterministic databases," in *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, vol. 00, Singapore, Singapore: IEEE, Nov. 2020, pp. 678–688. DOI: 10.1109/icdcs47774.2020.00040.

[60] E. P. C. Jones, D. J. Abadi, and S. Madden, "Low overhead concurrency control for partitioned main memory databases," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, ser. SIGMOD '10, Indianapolis, Indiana, USA: Association for Computing Machinery, Jun. 2010, pp. 603–614, ISBN: 9781450300322. DOI: 10.1145/1807167.1807233.

[61] R. Kallman, H. Kimura, J. Natkins, *et al.*, "H-store: A high-performance, distributed main memory transaction processing system," en, *Proceedings VLDB Endowment*, vol. 1, no. 2, pp. 1496–1499, Aug. 2008, ISSN: 2150-8097, 2150-8097. DOI: 10.14778/1454159.1454211.

[62] B. Kemme and G. Alonso, "Don't be lazy, be consistent: Postgres-R, a new way to implement database replication," in *Proceedings of the 26th International Conference on Very Large Data Bases*, ser. VLDB '00, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000, pp. 134–143. DOI: 10.5555/645926.671855.

[63] S. Kimball and I. Sharif, *Living without atomic clocks: Where CockroachDB and spanner diverge*, en, https://www.cockroachlabs.com/blog/living-without-atomic-clocks/, Accessed: 2024-7-25, Jan. 2022.

[64] H. F. Korth, S. Sudarshan, and A. Silberschatz, "Query optimization," en, in *Database System Concepts*, 6th ed., McGraw-Hill Education, Feb. 2019, ISBN: 9780078022159.

[65] P. Kraft, Q. Li, K. Kaffes, *et al.*, "Apiary: A DBMS-Integrated transactional Function-as-a-Service framework," Aug. 2022. arXiv: 2208.13068 [cs.DB].

[66] P. Kraft, Q. Li, X. Zhou, *et al.*, "Epoxy: ACID transactions across diverse data stores," en, *Proceedings VLDB Endowment*, vol. 16, no. 11, pp. 2742–2754, Jul. 2023, ISSN: 2150-8097. DOI: 10.14778/3611479.3611484.

[67] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete, "MDCC: Multi-data center consistency," in *Proceedings of the 8th ACM European Conference on Computer Systems*, ser. EuroSys '13, Prague, Czech Republic: Association for Computing Machinery, Apr. 2013, pp. 113–126, ISBN: 9781450319942. DOI: 10.1145/2465351.2465363.

[68] H. T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Transactions on Database Systems*, pp. 351–351, 1981. DOI: 10.1109/vldb.1979.718150.

[69] Z. Lai, C. Liu, and E. Lo, "When private blockchain meets deterministic database," *Proc. ACM SIGMOD Int. Conf. Manag. Data*, vol. 1, no. 1, pp. 1–28, May 2023, ISSN: 0730-8078. DOI: 10.1145/3588952.

[70] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," en, *Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010, ISSN: 0163-5980, 1943-586X. DOI: 10.1145/1773912.1773922.

[71] L. Lamport, "The part-time parliament," en, *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, May 1998, ISSN: 0734-2071, 1557-7333. DOI: 10.1145/279227.279229.

[72] L. Lamport, "Paxos made simple," *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pp. 51–58, Dec. 2001.

[73] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues, "Making geo-replicated systems fast as possible, consistent when necessary," in *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, ser. OSDI'12, Hollywood, CA, USA: USENIX Association, Oct. 2012, pp. 265–278, ISBN: 9781931971966. DOI: 10.5555/2387880.2387906.

[74] Q. Lin, P. Chang, G. Chen, B. C. Ooi, K.-L. Tan, and Z. Wang, "Towards a non-2PC transaction management in distributed database systems," in *Proceedings of the 2016 International Conference on Management of Data*, San Francisco California USA: ACM, Jun. 2016, pp. 1659–1674, ISBN: 9781450335317. DOI: 10.1145/2882903.2882923.

[75] S. Liu, L. Multazzu, H. Wei, and D. A. Basin, "NOC-NOC: Towards performance-optimal distributed transactions," *Proc. ACM SIGMOD Int. Conf. Manag. Data*, vol. 2, no. 1, pp. 1–25, Mar. 2024, ISSN: 0730-8078. DOI: 10.1145/3639264.

[76] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, Cascais Portugal: ACM, Oct. 2011, pp. 401–416, ISBN: 9781450309776. DOI: 10.1145/2043556.2043593.

[77] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Stronger semantics for low-latency geo-replicated storage," in *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, ser. nsdi'13, Lombard, IL: USENIX Association, Apr. 2013, pp. 313–328.

[78] H. Lu, S. Mu, S. Sen, and W. Lloyd, "NCC: Natural concurrency control for strictly serializable datastores by avoiding the timestamp-inversion pitfall," *Oper Syst Des Implement*, pp. 305–323, May 2023. DOI: 10.48550/arXiv.2305.14270.

[79] Y. Lu, X. Yu, L. Cao, and S. Madden, "Aria: A fast and practical deterministic OLTP database," en, *Proceedings VLDB Endowment*, vol. 13, no. 12, pp. 2047–2060, Aug. 2020, ISSN: 2150-8097. DOI: 10.14778/3407790.3407808.

[80] Y. Lu, X. Yu, L. Cao, and S. Madden, "Epoch-based commit and replication in distributed OLTP databases," en, *Proceedings VLDB Endowment*, vol. 14, no. 5, pp. 743–756, Jan. 2021, ISSN: 2150-8097. DOI: 10.14778/3446095.3446098.

[81] Y. Lu, X. Yu, and S. Madden, "STAR: Scaling transactions through asymmetric replication," *Proc. VLDB Endow.*, vol. 12, no. 11, pp. 1316–1329, Jul. 2019, ISSN: 2150-8097. DOI: 10.14778/3342263.3342270.

[82] H. Mahmoud, F. Nawab, A. Pucher, D. Agrawal, and A. El Abbadi, "Low-latency multi-datacenter databases using replicated commit," en, *Proceedings VLDB Endowment*, vol. 6, no. 9, pp. 661–672, Jul. 2013, ISSN: 2150-8097. DOI: 10.14778/2536360.2536366.

[83] P. Mattis, *Yugabyte vs. CockroachDB: Unpacking competitive benchmark claims*, en, https://www.cockroachlabs.com/blog/unpacking-competitive-benchmarks/, Accessed: 2024-7-12, Nov. 2019.

[84] C. Mohan, "ARIES/KVL: A Key-Value locking method for concurrency control of multiaction transactions operating on B-Tree indexes," *VLDB J.*, 1990, ISSN: 1066-8888. DOI: 10.5555/94362.94465.

[85] *Mongodb*, https://mongodb.com, 2009.

[86] I. Moraru, D. G. Andersen, and M. Kaminsky, "There is more consensus in egalitarian parliaments," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, Farminton Pennsylvania: ACM, Nov. 2013, pp. 358–372, ISBN: 9781450323888. DOI: 10.1145/2517349.2517350.

[87] S. Mu, Y. Cui, Y. Zhang, W. Lloyd, and J. Li, "Extracting more concurrency from distributed transactions," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14, Broomfield, CO: USENIX Association, 2014, pp. 479–494. DOI: 10.5555/2685048.2685086.

[88] S. Mu, L. Nelson, W. Lloyd, and J. Li, "Consolidating concurrency control and consensus for commits under conflicts," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 517–532. DOI: 10.5555/3026877.3026917.

[89] R. Murthy and G. Goindi, *AlloyDB for PostgreSQL intelligent scalable storage*, en, https://cloud.google.com/blog/products/databases/alloydb-for-postgresql-intelligent-scalable-storage, Accessed: 2024-2-3, May 2022.

[90] *Neon — serverless, Fault-Tolerant, branchable postgres*, en, https://neon.tech/, Accessed: 2024-2-3, 2021.

[91] C. D. T. Nguyen, J. K. Miller, and D. J. Abadi, "Detock: High performance multi-region transactions at scale," en, *Proc. ACM SIGMOD Int. Conf. Manag. Data*, vol. 1, no. 2, pp. 1–27, Jun. 2023, ISSN: 0730-8078. DOI: 10.1145/3589293.

[92] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," *USENIX*, 2014. DOI: 10.5555/2643634.2643666.

[93] Oracle, *Oracle RAC*, en, https://www.oracle.com/database/real-application-clusters/, Accessed: 2024-2-29, 2001.

[94] C. H. Papadimitriou, "The serializability of concurrent database updates," en, *J. ACM*, vol. 26, no. 4, pp. 631–653, Oct. 1979, ISSN: 0004-5411, 1557-735X. DOI: 10.1145/322154.322158.

[95] A. Pathak, H. Pucha, Y. Zhang, Y. C. Hu, and Z. M. Mao, "A measurement study of internet delay asymmetry," in *Lecture Notes in Computer Science*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 182–191, ISBN: 9783540792314. DOI: 10.1007/978-3-540-79232-1_19.

[96] A. Pavlo, "What are we doing with our lives? nobody cares about our concurrency control research," in *Proceedings of the 2017 ACM International Conference on Management of Data*, ser. SIGMOD '17, Chicago, Illinois, USA: Association for Computing Machinery, May 2017, p. 3, ISBN: 9781450341974. DOI: 10.1145/3035918.3056096.

[97] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers, "Flexible update propagation for weakly consistent replication," in *Proceedings of the sixteenth ACM symposium on Operating systems principles*, ser. SOSP '97, vol. 31, Saint Malo, France: Association for Computing Machinery, Oct. 1997, pp. 288–301, ISBN: 9780897919166. DOI: 10.1145/268998.266711.

[98] D. R. K. Ports and K. Grittner, "Serializable snapshot isolation in PostgreSQL," *Proceedings VLDB Endowment*, vol. 5, no. 12, pp. 1850–1861, 2012, ISSN: 2150-8097. DOI: 10.14778/2367502.2367523.

[99] G. Prasaad, A. Cheung, and D. Suciu, "Handling highly contended OLTP workloads using fast dynamic partitioning," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '20, Portland, OR, USA: Association for Computing Machinery, May 2020, pp. 527–542, ISBN: 9781450367356. DOI: 10.1145/3318464.3389764.

[100] S. Proctor, *Exploring the architecture of the nuodb database, part 1*, https://www.infoq.com/articles/nuodb-architecture-1, 2013.

[101] S. Proctor, *Exploring the architecture of the nuodb database, part 2*, https://www.infoq.com/articles/nuodb-architecture-2, 2013.

[102] *Production checklist | cockroachdb docs*, https://www.cockroachlabs.com/docs/stable/recommended-production-settings.htm, 2022.

[103] T. Qadah, S. Gupta, and M. Sadoghi, "Q-Store: Distributed, multi-partition transactions via queue-oriented execution and communication," in *EDBT*, Copenhagen, Denmark, 2020, pp. 73–84. DOI: 10.5441/002/edbt.2020.08.

[104] D. Qin, A. D. Brown, and A. Goel, "Caracal: Contention management with deterministic concurrency control," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP '21, Virtual Event, Germany: Association for Computing Machinery, Oct. 2021, pp. 180–194, ISBN: 9781450387095. DOI: 10.1145/3477132.3483591.

[105] S. Rahnama, S. Gupta, R. Sogani, D. Krishnan, and M. Sadoghi, "RingBFT: Resilient consensus over sharded ring topology," in *EDBT*, Edinburgh, UK: OpenProceedings.org, 2022, pp. 298–311. DOI: 10.48786/EDBT.2022.17.

[106] K. Ranganathan, *Bringing truth to competitive benchmark claims – YugabyteDB vs CockroachDB, part 1*, https://www.yugabyte.com/blog/yugabytedb-vs-cockroachdb-bringing-truth-to-performance-benchmark-claims-part-1/, Accessed: 2024-7-12, May 2020.

[107] K. Ranganathan, *Bringing truth to competitive benchmark claims – YugabyteDB vs CockroachDB, part 2*, https://www.yugabyte.com/blog/yugabytedb-vs-cockroachdb-bringing-truth-to-performance-benchmark-claims-part-2/, Accessed: 2024-7-12, May 2020.

[108] K. Ren, J. M. Faleiro, and D. J. Abadi, "Design principles for scaling multi-core OLTP under high contention," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16, San Francisco, California, USA: Association for Computing Machinery, Jun. 2016, pp. 1583–1598, ISBN: 9781450335317. DOI: 10.1145/2882903.2882958.

[109] K. Ren, D. Li, and D. J. Abadi, "SLOG: Serializable, low-latency, geo-replicated transactions," en, *Proceedings VLDB Endowment*, vol. 12, no. 11, pp. 1747–1761, Jul. 2019, ISSN: 2150-8097, 2150-8097. DOI: 10.14778/3342263.3342647.

[110] K. Ren, A. Thomson, and D. J. Abadi, "Lightweight locking for main memory database systems," en, *Proceedings VLDB Endowment*, vol. 6, no. 2, pp. 145–156, Dec. 2012, ISSN: 2150-8097. DOI: 10.14778/2535568.2448947.

[111] K. Ren, A. Thomson, and D. J. Abadi, "An evaluation of the advantages and disadvantages of deterministic database systems," en, *Proceedings VLDB Endowment*, vol. 7, no. 10, pp. 821–832, Jun. 2014, ISSN: 2150-8097. DOI: 10.14778/2732951.2732955.

[112] R. Schumacher, H. Chibber, and U. Dwivedi, *Market share: Database management systems, worldwide, 2022*, en, https://www.gartner.com/en/documents/4366299, Accessed: 2024-7-23.

[113] E. Schurman and J. Brutlag, "Performance related changes and their user impact," in *velocity web performance and operations conference*, 2009.

[114] Y. Sovran, R. Power, M. K. Aguilera, and J. Li, "Transactional storage for geo-replicated systems," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, Cascais Portugal: ACM, Oct. 2011, p. 385, ISBN: 9781450309776. DOI: 10.1145/2043556.2043592.

[115] M. Stonebraker, "The case for shared nothing," in *HPTS*, 1985.

[116] M. Stonebraker, G. Held, E. Wong, and P. Kreps, "The design and implementation of INGRES," *ACM Trans. Database Syst.*, vol. 1, no. 3, pp. 189–222, Sep. 1976, ISSN: 0362-5915. DOI: 10.1145/320473.320476.

[117] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland, "The end of an architectural era (it's time for a complete rewrite)," *VLDB J.*, 2007, ISSN: 1066-8888. DOI: 10.5555/1325851.1325981.

[118] R. Taft, I. Sharif, A. Matei, *et al.*, "CockroachDB: The resilient Geo-Distributed SQL database," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '20, Portland, OR, USA: Association for Computing Machinery, May 2020, pp. 1493–1509, ISBN: 9781450367356. DOI: 10.1145/3318464.3386134.

[119] C. Tang, Z. Wang, X. Zhang, *et al.*, "Ad hoc transactions in web applications: The good, the bad, and the ugly," in *Proceedings of the 2022 International Conference on Management of Data*, ser. SIGMOD '22, Philadelphia, PA, USA: Association for Computing Machinery, Jun. 2022, pp. 4–18, ISBN: 9781450392495. DOI: 10.1145/3514221.3526120.

[120] *Tc(8) — linux manual page*, https://man7.org/linux/man-pages/man8/tc.8.html, 2021.

[121] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser, "Managing update conflicts in bayou, a weakly connected replicated storage system," in *Proceedings of the fifteenth ACM symposium on Operating systems principles - SOSP '95*, vol. 29, Copper Mountain, Colorado, United States: ACM Press, 1995, pp. 172–182. DOI: 10.1145/224056.224070.

[122] A. Thomson and D. J. Abadi, "The case for determinism in database systems," en, *Proceedings VLDB Endowment*, vol. 3, no. 1-2, pp. 70–80, Sep. 2010, ISSN: 2150-8097, 2150-8097. DOI: 10.14778/1920841.1920855.

[123] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi, "Calvin: Fast distributed transactions for partitioned database systems," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '12, Scottsdale, Arizona, USA: Association for Computing Machinery, May 2012, pp. 1–12, ISBN: 9781450312479. DOI: 10.1145/2213836.2213838.

[124] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi, "Fast distributed transactions and strongly consistent replication for OLTP database systems," *ACM Trans. Database Syst.*, vol. 39, no. 2, pp. 1–39, May 2014, ISSN: 0362-5915. DOI: 10.1145/2556685.

[125] *TPC benchmark C*, en, https://www.tpc.org/tpcc/, Accessed: 2024-2-22, 2010.

[126] *Transparent application scaling with IBM DB2 purescale*, White Paper, Oct. 2019.

[127] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden, "Speedy transactions in multicore in-memory databases," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13, Farminton, Pennsylvania: Association for Computing Machinery, Nov. 2013, pp. 18–32, ISBN: 9781450323888. DOI: 10.1145/2517349.2522713.

[128] N. VanBenschoten, A. Ajmani, M. Gartner, *et al.*, "Enabling the next generation of Multi-Region applications with CockroachDB," in *Proceedings of the 2022 International Conference on Management of Data*, ser. SIGMOD '22, Philadelphia, PA, USA: Association for Computing Machinery, Jun. 2022, pp. 2312–2325, ISBN: 9781450392495. DOI: 10.1145/3514221.3526053.

[129] A. Verbitski, A. Gupta, D. Saha, *et al.*, "Amazon aurora: Design considerations for high throughput Cloud-Native relational databases," in *Proceedings of the 2017 ACM International Conference on Management of Data*, ser. SIGMOD '17, Chicago, Illinois, USA: Association for Computing Machinery, May 2017, pp. 1041–1052, ISBN: 9781450341974. DOI: 10.1145/3035918.3056101.

[130] A. Verbitski, A. Gupta, D. Saha, *et al.*, "Amazon aurora: On avoiding distributed consensus for I/Os, commits, and membership changes," in *Proceedings of the 2018 International Conference on Management of Data*, ser. SIGMOD '18, Houston, TX, USA: Association for Computing Machinery, May 2018, pp. 789–796, ISBN: 9781450347037. DOI: 10.1145/3183713.3196937.

[131] Z. Wang, S. Mu, Y. Cui, H. Yi, H. Chen, and J. Li, "Scaling multicore databases via constrained parallel execution," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16, San Francisco, California, USA: Association for Computing Machinery, Jun. 2016, pp. 1643–1658, ISBN: 9781450335317. DOI: 10.1145/2882903.2882934.

[132] T. Warszawski and P. Bailis, "ACIDRain: Concurrency-Related attacks on Database-Backed web applications," in *Proceedings of the 2017 ACM International Conference on Management of Data*, ser. SIGMOD '17, Chicago, Illinois, USA: Association for Computing Machinery, May 2017, pp. 5–20, ISBN: 9781450341974. DOI: 10.1145/3035918.3064037.

[133] *What is your ping, google cloud and amazon AWS?* en, https://www.bigbitbus.com/2018/05/07/What-Is-Your-Ping-AWS-And-Google-Cloud/, Accessed: 2024-7-23, May 2018.

[134] A. T. Whitney, D. Shasha, and S. Apter, "High volume transaction processing without currency control, two phase commit, SQL or c++," en, in *Seventh international workshop on high performance transaction systems*, Asimolar, California, 1997, pp. 211–217.

[135] Wikipedia contributors, *Separation of concerns*, https://en.wikipedia.org/wiki/Separation_of_concerns, Accessed: NA-NA-NA, Jun. 2024.

[136] X/Open Company Ltd., "Distributed transaction processing: The XA specification," Tech. Rep., 1991.

[137] X. Yan, L. Yang, and B. Wong, "Domino: Using network measurements to reduce state machine replication latency in WANs," in *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*, Barcelona Spain: ACM, Nov. 2020, ISBN: 9781450379489. DOI: 10.1145/3386367.3431291.

[138] X. Yan, L. Yang, H. Zhang, *et al.*, "Carousel: Low-Latency transaction processing for Globally-Distributed data," in *Proceedings of the 2018 International Conference on Management of Data*, ser. SIGMOD '18, Houston, TX, USA: Association for Computing Machinery, May 2018, pp. 231–243, ISBN: 9781450347037. DOI: 10.1145/3183713.3196912.

[139] L. Yang, X. Yan, and B. Wong, "Natto: Providing distributed transaction prioritization for High-Contention workloads," in *Proceedings of the 2022 International Conference on Management of Data*, ser. SIGMOD '22, Philadelphia, PA, USA: Association for Computing Machinery, Jun. 2022, pp. 715–729, ISBN: 9781450392495. DOI: 10.1145/3514221.3526161.

[140] X. Yang, Y. Zhang, H. Chen, C. Sun, F. Li, and W. Zhou, "PolarDB-SCC: A Cloud-Native database ensuring low latency for strongly consistent reads," *Proceedings VLDB Endowment*, vol. 16, no. 12, pp. 3754–3767, Aug. 2023, ISSN: 2150-8097. DOI: 10.14778/3611540.3611562.

[141] X. Yang, Y. Zhang, H. Chen, *et al.*, "PolarDB-MP: A Multi-Primary Cloud-Native database via disaggregated shared memory," in *Companion of the 2024 International Conference on Management of Data*, ser. SIGMOD/PODS '24, Santiago AA, Chile: Association for Computing Machinery, Jun. 2024, pp. 295–308, ISBN: 9798400704222. DOI: 10.1145/3626246.3653377.

[142] Z. Yang, C. Yang, F. Han, *et al.*, "OceanBase: A 707 million tpmc distributed relational database system," en, *Proceedings VLDB Endowment*, vol. 15, no. 12, pp. 3385–3397, Aug. 2022, ISSN: 2150-8097. DOI: 10.14778/3554821.3554830.

[143] C. Yao, D. Agrawal, G. Chen, *et al.*, "Exploiting single-threaded model in multi-core in-memory systems," *IEEE Trans. Knowl. Data Eng.*, vol. 28, no. 10, pp. 2635–2650, Oct. 2016, ISSN: 1041-4347, 1558-2191. DOI: 10.1109/tkde.2016.2578319.

[144] *YugabyteDB*, en, https://www.yugabyte.com/, Accessed: 2024-2-26, Nov. 2016.

[145] *YugabyteDB - deployment checklist*, en, https://docs.yugabyte.com/preview/deploy/checklist, Accessed: 2024-2-18, 2016.

[146] C. Yun, *Amazon aurora limitless database*, https://aws.amazon.com/blogs/aws/join-the-preview-amazon-aurora-limitless-database/, Accessed: 2024-2-29, Nov. 2023.

[147] E. Zamanian, J. Shun, C. Binnig, and T. Kraska, "Chiller: Contention-centric transaction execution and data partitioning for modern networks," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '20, Portland, OR, USA: Association for Computing Machinery, May 2020, pp. 511–526, ISBN: 9781450367356. DOI: 10.1145/3318464.3389724.

[148] *Zeromq*, https://zeromq.org/, 2007.

[149] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports, "Building consistent transactions with inconsistent replication," en, *ACM Trans. Comput. Syst.*, vol. 35, no. 4, pp. 1–37, Dec. 2018, ISSN: 0734-2071, 1557-7333. DOI: 10.1145/3269981.

[150] Z. Zhang, H. Hu, X. Zhou, and J. Wang, "Starry: Multi-master transaction processing on semi-leader architecture," *Proceedings VLDB Endowment*, vol. 16, no. 1, pp. 77–89, Sep. 2022, ISSN: 2150-8097. DOI: 10.14778/3561261.3561268.

[151] T. Ziegler, P. Bernstein, V. Leis, and C. Binnig, "Is scalable OLTP in the cloud a solved problem?" en, *Conference on Innovative Data Systems Research*, 2023.