HIGH-PERFORMANT REPLICATED QUEUE-ORIENTED TRANSACTION PROCESSING SYSTEMS ON MODERN COMPUTING INFRASTRUCTURES

by

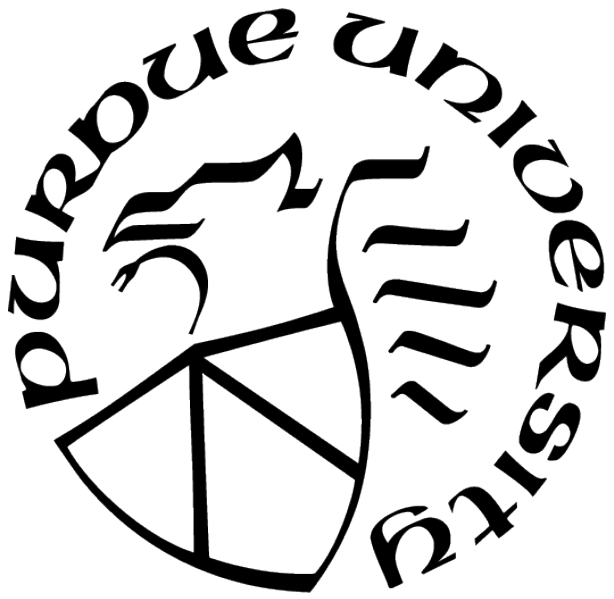
Thamir M. Qadah

A Dissertation

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Doctor of Philosophy



School of Electrical and Computer Engineering

West Lafayette, Indiana

August 2021

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. Arif Ghafoor, Co-Chair

School of Electrical and Computer Engineering, Purdue University

Dr. Mohammad Sadoghi, Co-Chair

Computer Science Department, University of California, Davis

Dr. Walid G. Aref

Department of Computer Science, Purdue University

Dr. Elisa Bertino

Department of Computer Science, Purdue University

Dr. Michael D. Zoltowski

School of Electrical and Computer Engineering, Purdue University

Approved by:

Dr. Dimitrios Peroulis

To my beloved parents Mohammad (Tajuddin) and Kalthom,
my wife Hanan, and my children Rayan and Iyad.

ACKNOWLEDGMENTS

First and foremost, all praise be to Allah Almighty, who blessed me and enabled me to complete my doctoral studies. The Ph.D. journey has been very stressful at times. However, the most vital source of my relief is the belief in the virtue and effectiveness of my *Istkhara* prayer prior to my decision to start my graduate studies at Purdue and my belief that it is the best choice made by Allah Almighty for me. Therefore, I ask Allah to continue guiding me to the right path and enabling me to become beneficial to others with the knowledge and skills I gained during my doctoral studies.

I am incredibly grateful to my advisor Prof. Mohammad Sadoghi for helping me in developing my research ideas and succeeding in publishing them in top peer-reviewed venues. His support in providing me with the computational resources to conduct the experiments in my dissertation is invaluable and crucial to my ability to perform extensive experimental evaluations of my research prototypes. Under his guidance, I was able to fulfill one of my goals and get the best paper award for a peer-reviewed paper as the lead author, which also happened to be my first! Working with him in the ExpoLab research group has been a wonderful and fulfilling experience. I am indebted to my advisor Prof. Arif Ghafoor for his continuous support during my studies. He has always been there to help me succeed in completing my doctoral studies through his invaluable advice and encouragement. In addition, I am grateful to Prof. Walid Aref for his excellent mentorship in the early stages of my Ph.D. journey and for exposing me to the joy of database systems research. Furthermore, I would like to extend my appreciation to Prof. Elisa Bertino and Prof. Micheal Zoltowski for serving on my doctoral committee and for their thoughtful and valuable feedback on my dissertation.

My utmost gratitude goes to my parents Mohammad (Tajuddin) Qadah and Kalthom Qadah, who endured my selfishness in pursuing my graduate studies in the United States. Their continuous prayers and encouragement for me have certainly played a big role in my success in completing this milestone.

I am probably will not find myself where I am now without the vital support from my wife, Hanan Alyamani, whose love, compassion, tolerance, patience, kindness, and sacrifices

have played a pivotal role in my ability to complete this journey. I am incredibly beholden to her for standing by and supporting me during my difficult times in this journey despite her own challenging situation being a student and a mother. Special thanks to my sons Rayan and Iyad for bringing me joy during my stressful times.

Also, I would like to extend my thanks to my father-in-law Prof. Ahmad Alyamani and mother-in-law Ensherah Makkawi whose prayers and encouragement have been a great source of confidence. Furthermore, I would like to thank my brothers, sisters, uncles, aunts for their encouragement and support. Many thanks to my colleagues from the Expolab research group and the Distributed Multimedia Systems lab for the intriguing discussions and their encouragement and friendship.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

With the shifting landscape of computing hardware architectures and the emergence of new computing environments (e.g., large main-memory systems, hundreds of CPUs, distributed and virtualized cloud-based resources), state-of-the-art designs of transaction processing systems that rely on conventional wisdom suffer from lost performance optimization opportunities. This dissertation challenges conventional wisdom to rethink the design and implementation of transaction processing systems for modern computing environments.

We start by tackling the vertical hardware scaling challenge, and propose a deterministic approach to transaction processing on emerging multi-sockets, many-core, shared memory architecture to harness its unprecedented available parallelism. Our proposed priority-based queue-oriented transaction processing architecture eliminates the transaction contention footprint and uses speculative execution to improve the throughput of centralized deterministic transaction processing systems. We build QueCC and demonstrate up to two orders of magnitude better performance over the state-of-the-art.

We further tackle the horizontal scaling challenge and propose a distributed queue-oriented transaction processing engine that relies on queue-oriented communication to eliminate the traditional overhead of commitment protocols for multi-partition transactions. We build Q-Store, and demonstrate up to 22x improvement in system throughput over the state-of-the-art deterministic transaction processing systems.

Finally, we propose a generalized framework for designing distributed and replicated deterministic transaction processing systems. We introduce the concept of speculative replication to hide the latency overhead of replication. We prototype the speculative replication protocol in QR-Store and perform an extensive experimental evaluation using standard benchmarks. We show that QR-Store can achieve a throughput of 1.9 million replicated transactions per second in under 200 milliseconds and a replication overhead of 8%-25% compared to non-replicated configurations.

1. INTRODUCTION

Transaction processing is an old-aged problem that has been an active area of research for the past 40 years[1]. Classical transaction processing is characterized as non-deterministic because the final database state cannot be entirely determined by the input database state and the input set of transactions. The output database state acceptable as long as the resulted history of concurrent transaction execution is equivalent to some serial history of execution according to serializability theory.

1.1 Motivations

The goal of transaction processing protocols is to ensure ACID properties and increase the concurrency of executed transactions. Serializable isolation ensures anomaly-free execution. Using other isolation levels (e.g., Read-committed) improves concurrency but is prone to producing anomalies that defy users' intentions and leave the database in an undesirable inconsistent state.

Due to the non-deterministic nature of classical transaction processing protocols, they suffer from performance issues on modern computing environments such as main-memory databases that use many-core and multi-socket CPUs, and cloud-based distributed environment.

With the shifting landscape of computing hardware architectures and the emergence of new computing environments (e.g., large main-memory systems, hundreds of CPUs, distributed and virtualized cloud-based resources), state-of-the-art designs of transaction processing systems that rely on conventional wisdom suffer from lost performance optimization opportunities. This dissertation challenges conventional wisdom to rethink the design and implementation of transaction processing systems for modern computing environments.

1.2 Dissertation Overview

In Chapter 2, we emerging multi-sockets, many-core, shared memory architecture to harness its unprecedented available parallelism. We propose a *queue-oriented, control-free*

concurrency architecture, referred to as QueCC, that exhibits minimal contention among concurrent threads by eliminating the overhead of concurrency control from the critical path of the transaction. Our proposed priority-based queue-oriented transaction processing architecture eliminates the transaction contention footprint and uses speculative execution to improve the throughput of centralized deterministic transaction processing systems. QueCC operates on batches of transactions in two deterministic phases of priority-based planning followed by control-free execution. We extensively evaluate our transaction execution architecture and compare its performance against seven state-of-the-art concurrency control protocols designed for in-memory stores. We demonstrate that QueCC can significantly outperform state-of-the-art concurrency control protocols under high-contention by up to 6.3×. Moreover, our results show that QueCC can process nearly 40 million YCSB transactional operations per second while maintaining serializability guarantees with write-intensive workloads. Remarkably, QueCC out-performs H-Store by up to two orders of magnitude.

In Chapter 3, we further tackle the horizontal scaling challenge of processing distributed transactions in distributed database systems. Distributed database systems partition the data across multiple nodes to improve the concurrency, which leads to higher throughput performance. Traditional concurrency control algorithms aim at producing an execution history equivalent to any serial history of transaction execution. Hence, an agreement on the final serial history is required for concurrent transaction execution. Traditional agreement protocols such as Two-Phase-Commit (2PC) are typically used but act as a significant bottleneck when processing distributed transactions that access many partitions. 2PC requires extensive coordination among the participating nodes to commit a transaction. Unlike traditional techniques, deterministic concurrency control techniques aim for producing an execution history that obeys a pre-determined transaction ordering.

Recent proposals for deterministic transaction processing demonstrate high potential for improving the system throughput, which had led to their successful commercial adoption. However, these proposals do not efficiently utilize and exploit modern computing resources and are limited by design to conservative execution.

We propose a novel distributed queue-oriented transaction processing paradigm that fundamentally re-thinks how deterministic transaction processing is performed. The proposed

paradigm supports multiple execution paradigms, multiple isolation levels, and is amenable to efficient resource utilization. We employ the principles of our proposed paradigm to build Q-Store, which is the first to support speculative execution and exploits intra-transaction parallelism efficiently among proposed deterministic and distributed transaction processing systems. We perform extensive evaluation against both deterministic and non-deterministic transaction processing protocols and demonstrate up to two orders of magnitude of improved performance.

Finally, in Chapter 4, we address the fault-tolerance and high availability challenge in deterministic database systems. Deterministic database systems have received increasing attention from the database research community in recent years. Despite their current limitations, recent proposals of distributed deterministic transaction processing systems demonstrated significant improvements over systems using traditional transaction processing techniques (e.g., two-phase-locking or optimistic concurrency control with two-phase-commit). However, the problem of ensuring high availability in deterministic distributed transaction processing systems has received less attention from the research community, and this aspect has not been analyzed and evaluated well. This chapter proposes a generic framework to model the replication process in deterministic transaction processing and apply it to study three cases from the literature. We design and implement QR-Store, a queue-oriented replicated transaction processing system, and extensively evaluate it with various workloads based on a transactional version of YCSB. Our prototype implementation QR-Store can achieve a throughput of 1.9 million replicated transactions per second in under 200 milliseconds and a replication overhead of 8%-25% compared to non-replicated configurations.

2. QUEUE-ORIENTED CONCURRENCY

An earlier version of this chapter appeared in [2]

2.1 Introduction

New multi-socket, many-core hardware architectures with tens or hundreds of cores are becoming commonplace in the market today [3]–[5]. This is a trend that is expected to increase exponentially, thus, reaching thousands of cores per box in the near future [6]. However, recent studies have shown that traditional transactional techniques that rely on extensive coordination among threads fail to scale on these emerging hardware architectures; thus, there is an urgent need to develop novel techniques to utilize the power of next generation of highly parallel modern hardware [7]–[11]. There is also a new wave to study deterministic concurrency techniques, e.g., the read and write sets are known *a priori*. These promising algorithms are motivated from the practical standpoint by examining the predefined stored procedures that are heavily deployed in customer settings [12]–[16]. However, many of the existing deterministic approaches do not fundamentally redesign their algorithms for the many-core architecture, which is the precise focus on this work, a novel deterministic concurrency control for modern highly parallel architectures.

The main challenge for transactional processing systems built on top of many-core hardware is the increased contention (due to increased parallelism) among many competing cores for shared resources, e.g., failure to acquire highly contended locks (pessimistic) or failure to validate contented tuples (optimistic). The role of concurrency control mechanisms in traditional databases is to determine the interleaving order of operations among concurrent transactions over shared data. But there is no fundamental reason to rely on concurrency control logic during the actual execution nor it is a necessity to force the same thread to be responsible for executing both transaction and concurrency control logic. This important realization has been observed in recent studies [10], [17] that may lead to a complete paradigm shift in how we think about transactions, but we have just scratched the surface. It is essential to note that the two tasks of establishing the order for accessing shared data and actually executing the transaction's logic are completely independent. Hence, these tasks can potentially be performed in different phases of execution by independent threads.

For instance, Ren et al. [17] propose ORTHRUS which operates based on pessimistic concurrency control, in which transaction executer threads delegate locking functionality to dedicated lock manager threads. Yao et al. [10] propose LADS that process batches of transactions by constructing a set of transaction dependency graphs and partition them into smaller pieces (e.g., min-cut algorithms) followed by dependency-graph-driven transaction execution. Both ORTHRUS and LADS rely on explicit message-passing to communicate among threads, which can introduce an unnecessary overhead to transaction execution despite the available shared memory model of a single machine. In contrast, QueCC embraces the shared memory model and applies determinism in a two-phase, priority-based, queue-oriented execution model.

The proposed work in this chapter is motivated by a simple profound question: *is it possible to have concurrent execution over shared data without having any concurrency control?* To answer this question, we investigate a deterministic approach to transaction processing geared towards multi-socket, many-core architectures. In particular, we propose QueCC, pronounced Quick, a novel queue-oriented, control-free concurrency architecture that exhibits minimal contention during execution and imposes no coordination among transactions while offering serializable guarantees. The key intuition behind our QueCC's design is to eliminate concurrency control by executing a set of batched transactions in two disjoint and deterministic phases of planning and execution, namely, decompose transactions into (predetermined) priority queues followed by priority-queue-oriented execution. In other words, we impose a deterministic plan of execution on batches of transactions, which eliminates the need for concurrency control during the actual execution of transactions.

2.1.1 Emergence of Deterministic Data Stores

Early proposals for deterministic execution for transaction processing aimed at data replication (e.g., [18], [19]). The second wave of proposals focused on deterministic execution in distributed environments, and lock-based approaches for concurrency control. For example, H-Store is exclusively tailored for partitionable workloads (e.g. [14]) as it essentially relies on partition-level locks and runs transactions serially within each partition. Calvin and all of its

derivatives primarily focused on developing a novel distributed protocol, where essentially all nodes partaking in distributed transactions execute batched transactions on all replicas in a predetermined order known to all. For local in-node concurrency, in Calvin all locks are acquired (in a pre-determined order to avoid deadlocks) before a transaction starts and if not all locks are granted, then the node stalls [12]. In fact, Calvin and QueCC dovetails, the former sequences transactions pre-execution to essentially (almost) eliminate agreement protocol while the latter introduces a novel predetermined prioritization and queue-oriented execution model to essentially (almost) eliminate the concurrency protocol.

Serializablility Deterministic data stores guarantee serializable execution of transactions seamlessly. A deterministic transaction processing engine needs to ensure that (a) the order of conflicting operations, and (b) the commitment ordering of transactions follow the same order that is determined prior to execution. With those two constraints are satisfied by the execution engine, serializable execution is guaranteed. In fact, from the scheduling point of view, deterministic data stores are less flexible compared to other serializable approaches [20], [21] because there is only one possible serial schedule that is produced by the execution engine. However, this allows the protocol to plan a near-optimal schedule that maximizes the throughput. Furthermore, given the deterministic execution, evaluating and testing the concurrency protocol is dramatically simplified because all non-determinism complexity has been eliminated. The determinism profoundly simplifies the recovery execution, in fact, normal and recovery routines become identical.

Future of Deterministic In-memory Data Stores Notably, deterministic data stores have their own advantages and disadvantages that they may not be optimal for every possible workload [21]. For instance, it is an open question how to support transactions that demands multiple rounds of back-and-forth client-server communication or how to support the traditional cursor-based accesses. Clients must register stored procedures in advance and supply all input parameters at run-time, i.e., the read-set and the write-set of a transaction must be known prior to execution, and the use of non-deterministic functions, e.g., `currentTime()`, is non-trivial. Notably, there have been several lightweight solutions to efficiently determining read/write (when not known as *a priori*) through a passive, pre-play execution model [12]–[16], [22].

2.1.2 Contributions

In this chapter, we make the following contributions:

- we present a rich formalism to model our re-thinking of how transactions are processed in QueCC. Our formalism does not suffer from the traditional data dependency conflicts among transactions because they are seamlessly eliminated by our execution model (Section 2.2).
- we propose an efficient deterministic, queue-oriented transaction execution model for highly parallel architectures, that is amenable to efficient pipelining and offers a flexible and adaptable thread-to-queue assignment to minimize coordination (Section 2.3).
- we design a novel two-phase, priority-based, queue-oriented planning and execution model that eliminates the need for concurrency control (Section 2.4).
- we prototype our proposed concurrency architecture within ExpoDB [23], [24], a comprehensive concurrency control testbed, which includes eight modern concurrency techniques, to demonstrate QueCC effectiveness compared to state-of-the-art approaches based on well-established benchmarks such as TPC-C and YCSB (Section 2.5).

2.2 Formalism

Before describing the design and architecture of QueCC, we first present data and transaction models used by QueCC.

2.2.1 Data Model

The data model used is the widely adopted key-value storage model. In this model, each record in the database is logically defined as a pair (k, v), where k uniquely identifies a record and v is the value of that record. Internally, we access records by knowing its physical record identifiers (RID), i.e., the physical address in either memory or disk.

Operations are modeled as two fundamental types of operations; namely, `READ` and `WRITE` operations. However, there are other kinds of operations such as `INSERT`, `UPDATE`, and `DELETE`. Those operations are treated as different forms of the `WRITE` operation[25].

2.2.2 Transaction Model

Transactions can be modeled as a DAG (Directed Acyclic Graphs) of "sub-transactions" called transaction fragments. Each fragment performs a sequence of operations on a set of records (each internally associated with a RID). In addition to the operations, each fragment is associated with a set of constraints that captures the application integrity. We formally define transaction fragments as follows:

Definition 2.2.1. ***(Transaction fragments):***
A transaction fragment f_{i} is defined as a pair (S_{op}, C), where S_{op} is a finite sequence of operations either `READ` or `WRITE` on records identified with RIDs that are mapped to the same contiguous RID range, and C is a finite set of constraints that must be satisfied post the fragment execution.

Fragments that belong to the same transaction can have two kinds of dependencies, and such dependencies are based on the transaction's logic. We refer to them as logic-induced dependencies, and they are of two types: (1) data dependencies and (2) commit dependencies [26]. Because these logic-induced dependencies may also exist among transaction fragments that belong to the same transaction, we call them intra-transaction dependencies to differentiate them from inter-transaction dependencies that exist between fragments that belong to different transactions. Inter-transaction dependencies are induced by the transaction execution model. Thus, they are also called execution-induced dependencies.

An intra-transaction data dependency between fragment f_{i}, and another fragment f_{j} such that f_{j} is data-dependent on f_{i} implies that f_{j} requires some data that is computed by f_{i}. To illustrate, consider a transaction that reads a value v_{i} of a particular record, say, r_{i} and updates the value v_{j} of another record, say, r_{j} such that $v_{\text{j}} = v_{\text{i}} + 1$. This transaction can be decomposed into two fragments f_{i}, and f_{j} with a data dependency between f_{i} and f_{j} such

that f_j depends on f_i. We formalize the notion of intra-transaction data dependencies as follows:

Definition 2.2.2. ***(Intra-transaction data dependency):***
An intra-transaction data dependency exist between two transaction fragments f_i and f_j, denoted as $f_i \xrightarrow{d} f_j$, if and only if both fragments belong to the same transaction and the logic of f_j requires data computed by the logic of f_i.

The second type of logic-induced dependency is called an intra-transaction commit dependency. This kind of dependency captures the atomicity of a transaction when some of its fragments may abort due to logic-induced aborts. We refer to such fragments as abortable fragments. Logic-induced aborts are the result of violating integrity constraints defined by applications, which are captured by the set of constraints C for each fragment. Intuitively, if a fragment is associated with at least one constraint that may not be satisfied post the execution of the fragment, then it is abortable.

A formal definition of abortable fragments is as follows:

Definition 2.2.3. ***(Abortable transaction fragments):***
A transaction fragment f_i is abortable if and only if $f_i.C \neq \phi$.

Using the definition of abortable fragments, intra-transaction commit dependencies are formally defined as follows:

Definition 2.2.4. ***(Intra-transaction commit dependency):***
An intra-transaction commit dependency exist between two transaction fragments f_i and f_j, denoted as $f_i \xrightarrow{c} f_j$, if and only if both fragments belong to the same transaction and f_i is abortable.

The notion of transaction fragments is similar in spirit to the notion of pieces [26]–[28], the notion of actions in DORA[11], and the notion of record actions in LADS [10]. However, unlike those notions, we impose a RID-range restriction on records (i.e., partitioned data) accessed by fragments and formally model the set of constraints associated with fragments.

Now, we can formally define transactions based on the fragments and their dependencies, as follows:

Definition 2.2.5. ***(Transactions):***

A transaction t_i *is defined as a directed acyclic graph (DAG)* $G_{t_i} := (V_{t_i}, E_{t_i})$*, where* V_{t_i} *is finite set of transaction fragments* $\{f_1, f_2, \ldots, f_k\}$*, and* $E_{t_i} = \{(f_p, f_q) | f_p \xrightarrow{d} f_q \vee f_p \xrightarrow{c} f_q\}$

In QueCC, there is a third type of dependencies that may exist between transaction fragments of *different* transactions, which are induced by the execution model. Therefore, they are called execution-induced dependencies. Since we are modeling transactions at the level of fragments, we capture them at that level. However, they are called "commit dependencies" in [29] when not considering the notion of transaction fragments. They are the result of speculative reading of uncommitted records [30]. We formally define them as follows:

Definition 2.2.6. ***(Inter-transaction commit dependency):***

An inter-transaction commit dependency exist between two transaction fragments f_i *and* f_j *is denoted as* $f_i \xrightarrow{s} f_j$*, if and only if both fragments belong to different transactions and* f_j *speculatively reads uncommitted data written by* f_i

Note that inter-transaction commit dependencies may cause cascading aborts among transactions. This problem can be mitigated by exploiting the idea of "early write visibility", which is proposed by Faleiro et al.[26].

Also, note that execution-induced data dependencies among transactions, used to model conflicts in traditional concurrency control mechanisms, are no longer possible in QueCC because these conflicts are seamlessly resolved and eliminated by the deterministic, priority-based, queue-oriented execution model of QueCC. Non-deterministic data stores that rely on traditional concurrency control mechanisms, suffer from non-deterministic aborts caused by their execution model that employs non-deterministic concurrency control. A notable observation is that deterministic stores eliminate non-deterministic aborts, which improves the efficiency of the transaction processing engine.

2.3 Priority-based, Queue-oriented Transaction Processing

We first offer a high-level description of our transaction processing architecture. Our proposed architecture (depicted in Figure 2.1) is geared towards a throughput-optimized in-memory stores.

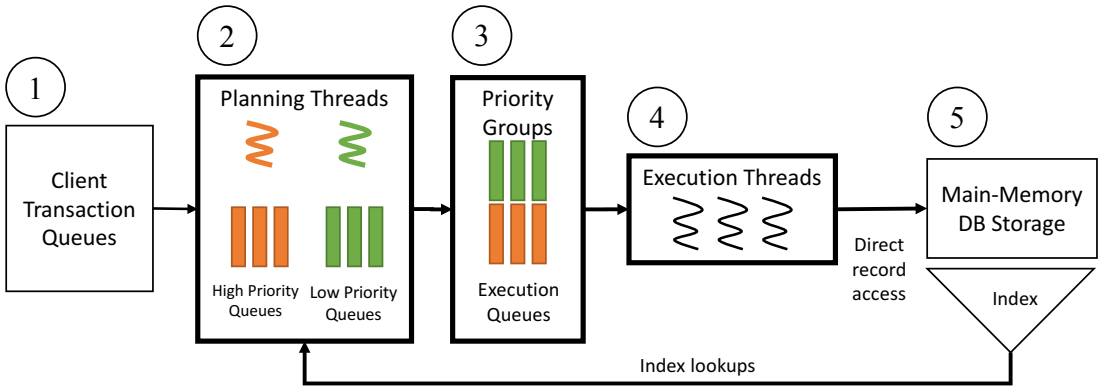


Figure 2.1. Overview of Priority-based, Queue-oriented Architecture

Transaction batches are processed in two deterministic phases. First, in the planning phase, multiple planner threads (②) consume transactions from their respective client transaction queue (①) in parallel and create prioritized execution queues (③). Each planner thread is assigned a predetermined distinct priority. The idea of priority is essential to the design of QueCC and it has two advantages. First, it allows planner threads to independently and in parallel perform their planning task. By assigning the priority to the execution queue, the ordering of transactions planned by different planner threads is preserved. Secondly, the priory enables execution threads to decide the order of executing fragments, which leads to correct serializable execution.

The planner thread acts as a local sequencer with a predetermined priority for its assigned transactions and spreads operations of each transaction (e.g., reads and writes) into a set of queues based on the sequence order.

Each queue is defined over a disjoint set of records, and queues inherit their planner distinct priorities. The goal of the planner is to distribute operations (e.g., `READ`/`WRITE`) into a set of almost equal-sized queues. Queues for each planner can be merged or split arbitrarily to satisfy balanced size queues. However, queues across planners can only be combined together following the strict priority order of each planner. We introduce *execution-priority invariance*, which is defined as follows:

Definition 2.3.1. ***(Execution-priority Invariance):***

For each record, operations that belong to higher priority queues (created by a higher priority planner) must always be executed before executing any lower priority operations.

The *execution-priority invariance* is the essence of how we capture determinism in QueCC. Since all planners operate at different priorities, then they can be plan independently in parallel without any contention.

The execution queues (③) are handed over to a set of execution threads (④) based on their priorities. An execution thread can arbitrarily select any outstanding queues within a batch and execute its operations without any coordination with others executors. The only criterion that must be satisfied is the *execution-priority invariance*, implying that if a lower priority queue overlaps with any higher priority queues (i.e., containing overlapping records), then before executing a lower priority queue, the operations in all higher priority queues must be executed first. Depending on the number of operations per transaction and its access patterns, independent operations from a single transaction may be processed in parallel by multiple execution threads without any synchronization among the executors; hence, coordination-free and independent execution across transactions. Execution threads operate directly on the in-memory store (⑤). Once all the execution queues are processed, it signals the completion of the batch, and transactions in the batch are committed except those that violated an integrity constraint. The violations are identified by executing a set of commit threads once each batch is completed.

To ensure recoverability, all parameters required to recreate the execution queues are persisted at the end of the planning phase. A second persistent operation is done at the end of the execution phase once the batch is fully processed; which is similar to the group commit technique [31].

2.3.1 Proof of serializability

In this section, we show that QueCC produces serializable execution histories. We use $c(T_i)$ to denote the commit ordering of transaction T_i, and $e(f_{ij})$ to denote the completion time for the execution of fragment f_{ij}, where f_{ij} belongs to T_i. For the sake of this proof, we

use the notion of conflicting fragments to have the same meaning as conflicting operations in serializability theory [32]. Without loss of generality, we assume that each fragment accesses a single record, but the same argument applies in general because of the RID range restriction (see Definition 2.2.1).

Theorem 2.3.1. *The transaction execution history produced by* QueCC *is serializable.*

Proof. Suppose that the execution of two transactions T_i and T_j is not serial, and their commit ordering is $c(T_i) < c(T_j)$. Note that their commitment ordering is the same as their ordering when they were planned. Therefore, there exist two conflicting fragments f_{ip} and f_{jq} such that $e(f_{jq}) > e(f_{ip})$. Because f_{ip} and f_{jq} access the *same* record, we have the following cases: (Case 1) if T_i and T_j are planned by the same planner thread, they must be placed in the *same* execution queue (EQ). Since the commitment ordering is the same as the order they were planned, the planner must have placed f_{ip} ahead of f_{jq} in the execution queue which contradicts the conflicting order. (Case 2) if T_i and T_j are planned by *different* planner threads, their respective fragments are placed in two different EQs with the EQ containing f_{ip} having a higher priority than the other EQ containing f_{jq}. Having $e(f_{jq}) > e(f_{ip})$ implies that the priority execution invariance is violated, which is also a contradiction. □

2.4 Control-free Architectural Design

In this section, we present planning and execution techniques introduced by QueCC.

2.4.1 Deterministic Planning Phase

In the planning phase, our aim is to answer the key questions: *how to efficiently produce execution plans and distribute them across execution threads in a balanced manner? How to efficiently deliver the plans to execution threads?*

A planner thread consumes transactions from its dedicated client transaction queue , which eliminates contention from using a single client transaction queue. Since each planner thread has its own pre-determined priority, at this point, transactions are partially ordered based the planners' priorities. Each planner can independently determine the order within

its own partition of the batch. The set of execution queues (EQs) filled by planners inherit their planner's priority thus forming a priority group (PG) of EQs (③). To represent priority inheritance of EQs, we associate all EQs planned by a planner with a priority group (PG). Each batch is organized into priority groups of EQs with each group inheriting the priority of its planner. We formally define the notion of a priority group as follows:

Definition 2.4.1. ***(Priority Group):***
Given a set of transactions in a batch, $T = \{t_1, t_2, \ldots, t_n\}$*, and a set of planner threads* $\{pt_1, pt_2, \ldots . pt_k\}$*, the planning phase will produce a set of k priority groups* $\{pg_1, pg_2, \ldots pg_k\}$*, where each* pg_{i} *is a partition of* T *and is produced by planner thread* pt_{i}*.*

In QueCC, EQs are the main data structure used to represents the workload of transaction fragments. Planners fill EQs with transaction fragments augmented with some additional meta-data during the planning and assign EQs to execution threads on batch delivery. EQs are recycled across batches, and they are dynamically expanded to hold transaction fragments beyond their initial capacity. Planners may physically split or logically merge EQs in order to balance the load given to execution queues. Splitting EQs is costly because it requires copying transaction fragments from one queue to two new queues that resulted from the split. The cost of allocating memory for EQs is minimized by maintaining a thread-local pool of EQs, which allows recycling EQs after batch commitment.

We now focus on how each planner produces the priority-based EQs associated with its PG. Our planning technique is based on RID value ranges.

Range-based Planning In our range-based planning approach, each planner starts by partitioning the whole RID space into a number of ranges equal to the number of execution threads[1]. For example, if we have 4 execution threads, then we will initially have 4 range partitions of the whole RID space. Based on the number of transactions accessing each range, that range can be further partitioned progressively into smaller ranges to ensure that they can be assigned to execution threads in a balanced manner (i.e., each execution thread will have the same number of transaction fragments to process). Note that each range is associated with an EQ, and partitioning a range implies splitting their associated EQs as

[1]↑The range partitioning can be learned, adapted, and tuned across batches

well. Range partitioning from earlier batches is reused for future ones, which amortize the cost of range partitioning across multiple batches, and reduces the planning time for the subsequent batches.

A range needs to be partitioned if its associated EQ is full. In QueCC, we have an adaptable system configuration parameter that controls the capacity of EQs. When EQs become full during planning, they are split into additional queues. The split algorithm is simple. Given an EQ to split, a planner partitions its associated range in half. Each range split will be associated with a new EQ obtained from a local thread pool of preallocated EQs[2]. Based on the new ranges, planners copy transaction fragments from the original EQ into the two new EQs.

A planner needs to determine when a batch is ready. Batches can be considered complete based on time (i.e., complete a batch every 5 milliseconds) or based on counts (i.e., complete a batch every 1000 transaction). The choice of how batches are determined is orthogonal to our techniques. However, in our implementation, we use count-based batches with the batch size being a configurable system parameter. Using count-based batches allows us to easily study the impact of batching. For count-based batches, a planner thread can easily compute the number of transactions in its partition of the batch since the number of planners and the batch size, are known parameters. Once the batch is planned and ready, it can be delivered to execution threads for execution.

Operation Planning Planning `READ` and `UPDATE` operations are straightforward, but special handling is needed for planning `INSERT` operations. When planning a `READ` or an `UPDATE` operation, a planner will simply do an index lookup to find the RID value for the record and its pointer. Based on the RID value, it determines the EQ responsible for the transaction fragment. It will check if the EQ is full and perform a split if needed. Finally, it inserts the transaction fragment into the EQ. `DELETE` operations are handled the same way as `UPDATE` operations from planning perspective. For the `INSERT` operations, a planner assigns a new RID value to the new record and places the fragment into the respective EQ.

[2]↑If the pool is empty, a new EQ is dynamically allocated.

2.4.2 Deterministic Execution Phase

Once the batch is delivered, execution threads start processing transaction fragments from assigned EQs without any need for controlling its access to records. Fragments are executed in the same order they are planned within a single EQ. Execution threads try to execute the whole EQ before moving to the next EQ. The execution threads may encounter a transaction fragment that has an intra-transaction data dependency to another fragment that resides in another EQ. Data dependencies exist when intermediate values are required to execute the fragment in hand. Once the intermediate values are computed by the corresponding fragments, they are stored in the transaction's meta-data accessible by all transaction fragments. Data dependencies may trigger EQ switching before the whole EQ is consumed. In particular, an EQ switch occurs if intermediate values required by the fragment in hand are not available.

To illustrate, consider the example transaction from Section 2.2, which has the following logic: $f_{\text{i}} = \{a = read(k_{\text{i}})\}, f_{\text{j}} = \{b = a+1; write(k_{\text{j}}, b)\}$, where keys are denoted as k_{i}. In this transaction, we have a data dependency between the two transaction fragments. The `WRITE` operation on k_{j} cannot be performed until the `READ` operation on k_{i} is completed. Suppose that f_{i} and f_{j} are placed in two separate EQs, e.g., EQ_1 and EQ_2 respectively. An attempt to execute f_{i} before f_{j} can happen, which triggers an EQ switch by the attempting execution thread. Note that, this delaying behavior[3] is unavoidable because there is no way for f_{j} to complete without the completion of f_{i}. This mechanism of EQ switching ensures that the execution thread only waits if data dependencies associated with transaction fragments at the head of all EQs are not satisfied. Our EQ switch mechanism is very lightweight and requires only a single private counter per EQ to keep track of how many fragments of the EQ are consumed.

Execution Priority Invariance Each execution thread (ET) is assigned one or more EQ in each PG. ETs can execute fragments from multiple PGs. Since EQs are planned independently by each planner, the following degenerate case may occur. Consider two

[3]↑Notably, although further processing of a queue maybe delayed, the executor is not blocked and may simply begin processing another queue.

planner threads, say, PT_0 and PT_1 with their respective PGs (i.e., PG_0 and PG_1), and two execution threads ET_0 and ET_1. A total of four EQs are planned in the batch. Each EQ is denoted as EQ_{ij} such that i refers to the planner thread index and j refers to the execution thread index, according to the assignment. For example, EQ_{00} is assigned by PT_0 to ET_0, and so forth. Therefore, we have the following set of EQs: $EQ_{00}, EQ_{01}, EQ_{10}$, and EQ_{11}. Now for each EQ, there is an associated RID range r_{ij}, and the indices of the ranges correspond to planner and execution threads, respectively. A violation of the *execution priority invariance* occurs under the following conditions: (1) ET_0 start executing EQ_{10}; (2) ET_1 has not completed the execution of EQ_{01}; (3) a fragment in EQ_{01} updates a record, while a fragment in EQ_{10} reads the same record (this implies that r_{10} overlaps with r_{01}). Therefore, to ensure the invariance, an executor checks that all overlapped EQs from higher priority PGs have completed their processing. If so, it proceeds with the execution of the EQ in hand, otherwise, it switches to another EQ. Fully processing all planned EQs in a batch signifies that all transactions are executed, and execution threads can start the commit stage for the whole batch. Notably, at any point during the execution, the executor thread may act as commit thread, by checking commit dependencies of fully executed transactions as described next.

Commit Dependency Tracking When processing a transaction, execution threads need to track inter-transaction commit dependencies. When a transaction fragment speculatively reads uncommitted data written by a fragment that belongs to another transaction in the batch, a commit dependency is formed between the two transactions. This dependency must be checked during commitment (or as soon as all prior transactions are fully executed) to ensure that the earlier transaction has committed. If the earlier transaction is aborted, the later transaction must abort. This dependency information is stored in the transaction context. To capture such dependencies, QueCC uses a similar approach to the approach used in [29], [30] for dealing with commit dependencies. QueCC maintains the transaction id of the last transaction that updated a record in per-record meta-data. During execution, the transaction ID is checked and if it refers to a transaction that belongs to the current batch, a commit dependency counter for the current transaction is incremented and a pointer to the current transaction's context is added to the context of the other transaction. During the

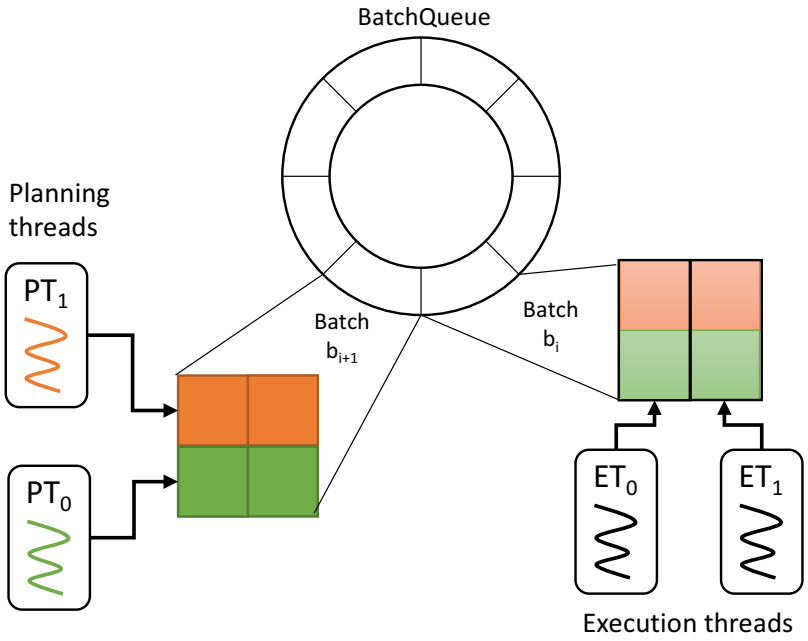


Figure 2.2. Example of concurrent batch planning and execution with 4 worker threads (2 planner threads + 2 execution threads). Priority groups are color-coded by planners. Execution threads process transactions from both priority groups.

commit stage, when a transaction is committing, the counters for all dependent transactions is decremented. When the commit dependency counter is equal to zero, the transaction is allowed to commit. Once all execution threads are done with their assigned work, the batch goes through a commit stage. This can be done in parallel by multiple threads.

2.4.3 QueCC Implementation Details

Plan Delivery After each planner, completes its batch partition and construct its PG, it needs to be delivered to the execution layer so that execution threads can start executing transactions. In QueCC, we use a simple lock-free delivery mechanism using atomic operations. We utilize a shared data structure called *BatchQueue*, which is basically a circular buffer that contains slots for each batch. Each batch slot contains pointers to partitions of priority groups which are set in a latch-free manner using atomic CAS operations. Priority group partitions are assigned to execution threads. Figure 2.2, illustrates an example of concurrent batch planning and execution of batch b_{i+1} and b_i respectively. In this example,

planner threads denoted as PT_0 and PT_1 are planning their respective priority groups for batch b_{i+1}; and concurrently, execution threads ET_0 and ET_1 are executing EQs from the previously planned batch (i.e., batch b_i).

Delivering priority group partitions to the execution layer must be efficient and lightweight. For this reason, QueCC uses a latch-free mechanism for delivery. The mechanism goes as follows. Execution threads spin on priority group partition slots while they are not set (i.e., their values is zero). Once the priority groups are ready to be delivered, planner threads merge EQs into priority group partitions such that the workload is balanced, and each priority group partition is assigned to one execution thread. EQs can be assigned dynamically by adapting to the workload or deterministically. To achieve balanced workload among execution threads, we have a simple greedy algorithm that keeps track of how many transaction fragments are assigned to each execution thread. It iterates over the remaining unassigned EQs until all EQs are assigned. In each iteration, it assigns an EQ to the worker with the lowest load.

Once a planner is done with creating execution threads assignments, it uses atomic CAS operations to set the values of the slots in the *BatchQueue* to point to the list of assigned EQs for each execution thread, which constitutes the priority group partition assigned to the respective execution thread.

Planning and Execution phases can be pipelinedin QueCC. In the pipelined design, execution threads are either processing EQs or waiting for their slots to be set by planner threads. As soon as the slot is set, execution threads can start processing EQs from the newly planned batch. On the other hand, for the un-pipelined design, worker threads acting as planner threads, will synchronize at the end of the planning phase. Once the synchronization is completed, worker threads will act as execution threads and start executing EQs.

Note that in QueCC, regardless of the number of planner threads and execution threads, there is zero contention with respect to the *BatchQueue* data structure.

RID Management Our planning is based on record identifiers (RIDs). RIDs can be physical or logical depending on the storage architecture being row-oriented or column-oriented. Typically, in row-oriented storage, physical RIDs are used. While in column-oriented storage, logical RIDs are used. As opposed to traditional disk-oriented data stores,

where RIDs are typically physical and is composed of the disk page identifier and the record offset, main-memory stores typically uses memory pointers as physical RIDs. On the other hand, logical RIDs are independent of the storage layout. Therefore, they can facilitate planning tasks since planners are dynamically creating logical partitions of the database by planning EQs. Logical RIDs leads to performance improvements when a set of independently accessed records are re-clustered logically regardless of their physical clustering. In QueCC, we use logical RIDs from a single space of 64-bit integers and are stored alongside index entries.

2.4.4 Discussion

QueCC supports "speculative write visibility" (SWV) when executing transaction fragments because it defers commitment to the end of the batch and allows reading uncommitted data written within a batch. In general, transaction fragments that may abort can cause cascading aborts. To ensure recoverability, QueCC maintains an undo buffer per transaction, which is populated by the before-image of records (or fields) being updated. A transaction can abort only if at least one of its fragments is abortable and have exercised its abort action.

If a transaction aborts, the original values are recovered from the undo buffers. This approach makes conservative assumptions about the *abortability* of transaction fragments (i.e., it assumes that all transaction fragments can abort). The overhead of maintaining undo-buffers can be eliminated if the transaction fragment is guaranteed to commit (i.e., it does not depend on other fragments). We can maintain information on the *abortability* of a transaction fragment in its respective transaction meta-data. Thus, instead of performing populating the undo buffers "blindly", we can check the possibility of an abort by looking at the transaction context, and skip the copying to undo buffers if the transaction is guaranteed to commit (i.e., passed its commit point[26]).

However, QueCC is not limited to only SWV and can support multiple write visibility policies. Faleiro et al. [26] introduced a new write visibility policy called "early write visibility" (EWV), which can improve the throughput of transaction processing by allowing reads on records only if their respective writes are guaranteed to be committed with serializability

Table 2.1. YCSB Workload configurations. *Notes: default values are in parenthesis; in partitioned stores, it reflects the number of partitions; batch size parameters are applicable only to* QueCC*; multi-partition transaction parameter is applicable only to the partitioned stores.*

Parameter Name	**Parameter Values**
# of worker threads	$4, 8, 16, 24, (32)$
Zipfian's theta	$0.0, 0.4, 0.8, 0.9, (0.99)$
% of write operations	$0\%, 5\%, 20\%, (50\%), 80\%, 95\%$
Rec. sizes	$50B, (100B), 200B, 400B, 800B, 1KB, 2KB$
Operations per txn	$1, 10, (16), 20, 32$
Batch sizes	$1K, 4K, (10K), 20K, 40K, 80K$
% of multi-partition txns.	$1\%, 5\%, 10\%, 20\%, 50\%, 80\%, 100\%$

guarantees. Unlike SWV, which is prone to cascading aborts, EWV is not. In fact, both EWV and SWV can be used at the same time by QueCC. A special token is placed ahead of the original fragment to make ETs adhere to the EWV policy. If that special token is not placed, then ETs will follow SWV course. One major advantage of using EWV in the context of QueCC is eliminating the process of backing-up copies of records in the undo-buffers. Since the transaction that updated record is guaranteed to commit, there will be no potential rollback and the undo-action is unnecessary.

2.5 Experimental Analysis

We have evaluated the QueCC protocol in our ExpoDB platform [23], [24]. ExpoDB is an in-memory, distributed transactional framework that not only offers a testbed to study concurrency and agreement protocols but also has a secure transactional capability to study distributed ledger—blockchain. ExpoDB's comprehensive concurrency testbed includes a variant of two-phase locking [33] (i.e., NO-WAIT [34] as a representative of pessimistic concurrency control), TicToc [35], Cicada [9], SILO [8], FOEDUS with MOCC [36], ERMIA with SSI and SSN [37], and H-Store [14], all of which were compared against QueCC.

2.5.1 Experimental Setup

We run all of our experiments using a Microsoft Azure G5 VM instance. This VM is equipped with an Intel Xeon CPU E5-2698B v3 running at 2GHz, and has 32 cores. The memory hierarchy includes a 32KB L1 data cache, 32KB L2 instruction cache, 256KB L2 cache, 40MB L3 cache, and 448GB of RAM. The operating system is Ubuntu 16.04.3 LTS (xenial). The codebase is compiled with GCC version 5.4.0 and $-O3$ compiler optimization flag.

The workloads are generated at the server before any transaction is processed, and are stored in main-memory buffers. This is done to remove any effects of the network, and allows us to study concurrency control protocols under high stress.

Every experiment starts with a warm-up period where measurements are not collected; followed by a measured period. Each experiment is run three times, and the average value is reported in the results of this section.

We focus on evaluating three metrics: throughput, latency, and abort percentage. The abort percentage is computed as the ratio between the total number of aborted transaction to the sum of the total number of attempted transaction (i.e., both aborted and committed transactions).

2.5.2 Workloads Overview

We have experimented with both YCSB and TPC-C benchmarks. Below, we briefly discuss the workloads used in our evaluation.

YCSB[38] is a web-application benchmark that is representative of web applications used by YAHOO. While the original workload does not have any transaction semantics, ours is adapted to have transactional capability by including multiple operations per transaction. Each operation can be either a `READ` or a `READ-MODIFY-WRITE` operation. The ratio of `READ` to `WRITE` operations can also vary. The benchmark consists of a single table. The table in our experiments contains 16 million records. Table 2.1 summarizes the various configuration parameters used in our evaluation, and default values are in parenthesis. The data access patterns can be controlled using the parameter θ of the Zipfian distribution. For example,

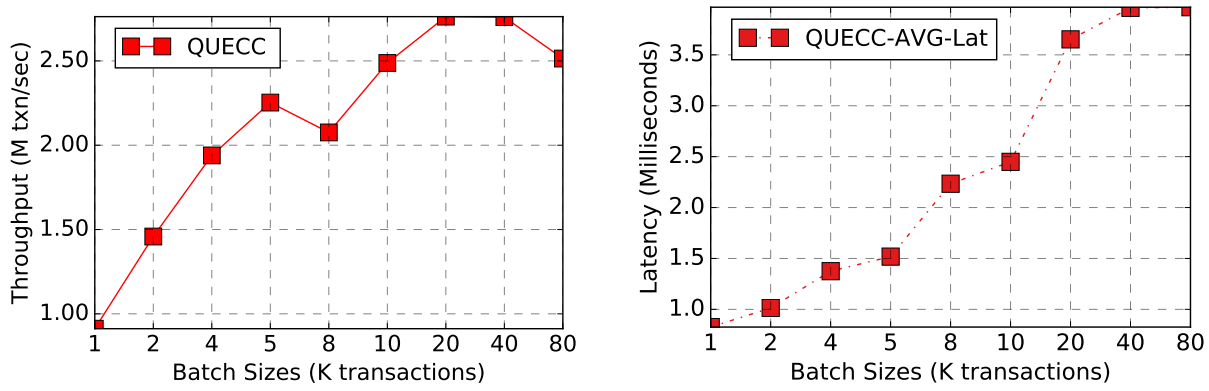
Table 2.2. TPC-C Workload configurations, default values are in parenthesis

Parameter Name	Parameter Values
# of worker threads	$4, 8, 16, 24, (32)$
% of payment txns.	$0\%, 50\%, 100\%$

a workload with uniform access has $\theta = 0.0$, while a skewed workload has a larger value of theta e.g., $\theta = 0.99$.

TPC-C [39] is the industry standard benchmark for evaluating transaction processing systems. It basically simulates a wholesale order processing system. Each warehouse is considered to be a single partition. There are 9 tables and 5 transaction types for this benchmark. The data store is partitioned by warehouse, which is considered the best possible partitioning scheme for the TPC-C workload [40]. Similar to previous studies in the literature[7], [13], we focus on the two main transaction profiles (`NewOrder` and `Payment`) out of the five profiles, which correspond to 88% of the default TPC-C workload mix. These two profiles are also the most complex ones. For example, the `NewOrder` transaction performs 2 `READ` operations, $6 - 16$ `READ-MODIFY-WRITE` operations, $7 - 16$ `INSERT` operations, and about 15% of these operations can access a remote partition. The *Payment* transaction, on the other hand, performs 3 `READ-MODIFY-WRITE` operations, and 1 `INSERT` operation. One of the reads uses the last name of the customer, which requires a little more work to look up the record.

In this chapter, we primarily study high-contention workloads because when there is limited or no contention, then, generally, the top approaches behave comparably with negligible differences. This choice also has an important practical significance [8], [9], [29], [35], [36] because real workloads are often skewed, thus, exhibiting a high contention. Therefore, in the interest of space, we present our detailed results for high-contention workloads and briefly overview the results for lower-contention scenarios.



(a) Throughput (b) Latency

Figure 2.3. Varying batch sizes and high data access skew ($\theta = 0.99$)

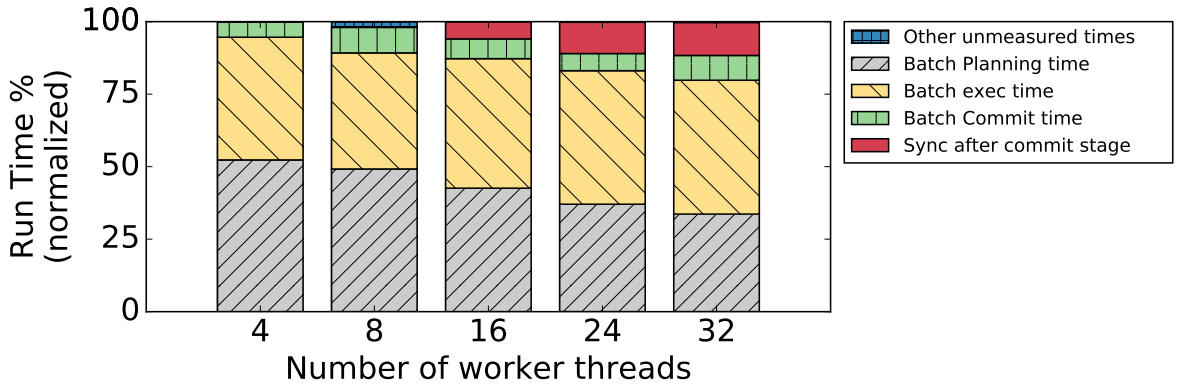


Figure 2.4. Time breakdown when varying number of worker threads.

2.5.3 YCSB Experiments

Using YCSB workloads, we start by evaluating the performance of QueCC with different batch sizes, which is a unique aspect of QueCC. Subsequently, we compare QueCC with other concurrency control protocols.

Effect of Batch Sizes We gear our experiments to study the effect of batch sizes on throughput and latency for QueCC because it is the only approach that uses batching. We use a write-intensive workload, 32 worker threads, a record size of 100 bytes, Zipfian's $\theta = 0.99$, and 16 operations per transaction.

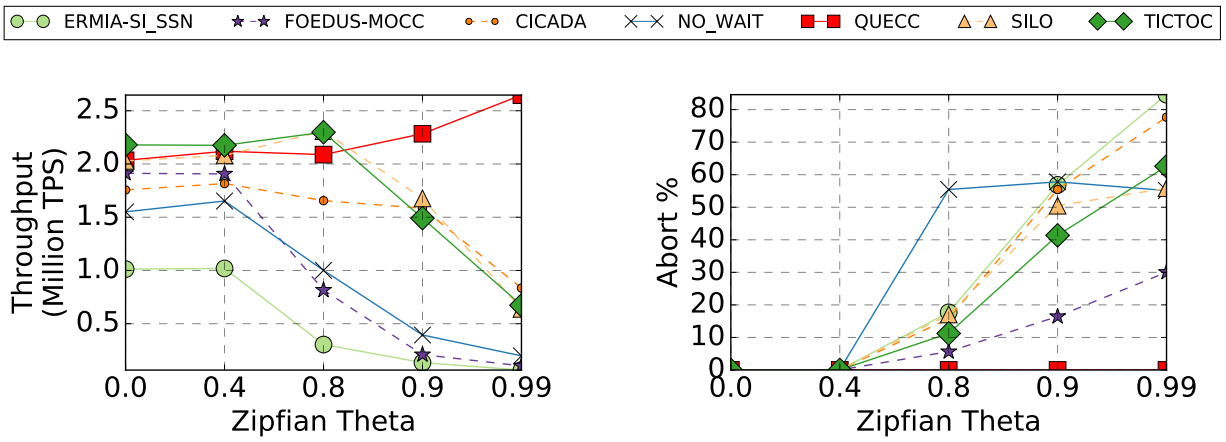
We observe that QueCC exhibit low average latency (i.e., under 3ms) for batches smaller than $20K$ transactions Figure 2.3b, which is considered reasonable for many applications. For the remaining experiments, we use a batch of size $10K$.

Time Breakdown Figure 2.4 illustrates the time breakdown spent on each phase of QueCC under highly skewed data accesses. Notably, QueCC continues to achieve high-utilization even under extreme contention model. For example, even scaling to 32 worker threads, over the 80% of the time is dedicated to useful work, i.e., planning and execution phases.

Effect of Data Access Skew We evaluate the effect of varying record contention using Zipfian's θ parameter of the YCSB workload while keeping the number of worker threads constant. We use 32 worker threads and assign one to each available core. Figure 2.5a, shows the throughput results of QueCC compared with other concurrency control protocols. We use a write-intensive workload which has 50% `READ-MODIFY-WRITE` operations per transaction. As expected QueCC performs comparably with the best competing approaches under low contention scenarios $\theta <= 0.8$. Remarkably, in high contention scenarios, QueCC begins to significantly outperforms all the state-of-the-art approaches. QueCC improves the next best approach by 3.3× with $\theta = 0.99$, and has 35% better throughput with $\theta = 0.9$. The main reason for QueCC's high-throughput is that it eliminates concurrency control induced aborts completely. On the other hand, the other approaches suffer from excessive transaction aborts which lead to wasted computations and complete stalls for lock-based approaches. This experiments also highlights the stability and predictability of QueCC with respect to degree of contention.

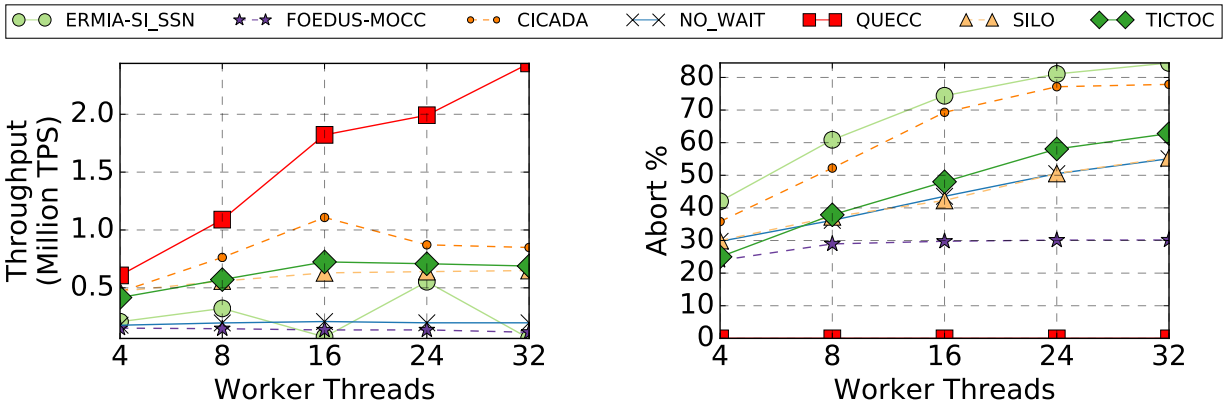
Scalability We evaluate the scalability of QueCC by varying the number of worker threads while maintaining a skewed, write-intensive access pattern. We observe that all other approaches scale poorly under highly concurrent access scenario (2.6a) despite employing techniques to reduce the cost of contention (e.g., Cicada). In contrast, QueCC scales well despite the higher contention due to increased number of threads. For instance, QueCC achieves nearly 3× the throughput of Cicada with 32 worker threads.

This result demonstrates the effectiveness of QueCC's concurrency architecture that exploits the untapped parallelism available in transactional workloads. Figure 2.6b shows that



(a) Throughput (b) Abort Percentage

Figure 2.5. Variable contention (θ) on write-intensive YCSB workload

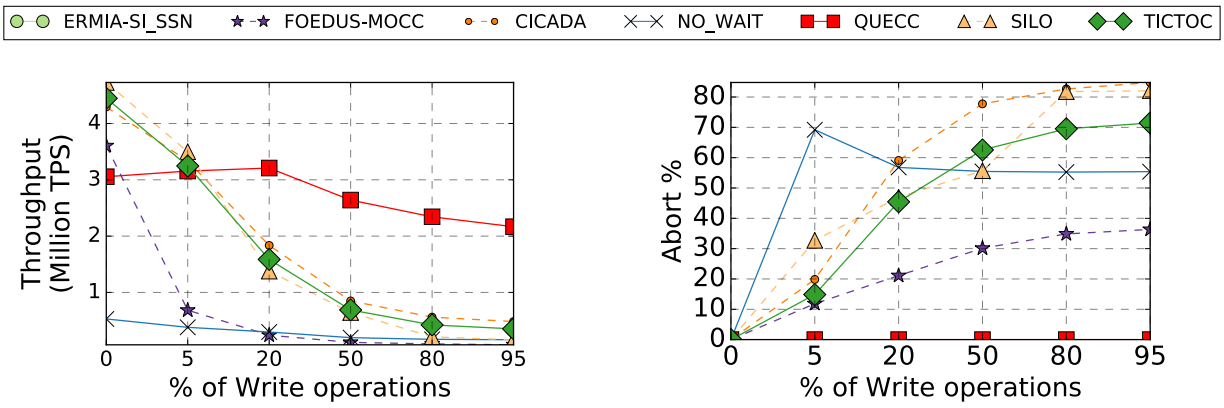


(a) Throughput (b) Abort percentage

Figure 2.6. Scaling Worker Threads Under Write Intensive Workload. High contention, $\theta = 0.99$

the abort rate for Cicada, TicToc, and ERMIA as parallelism increases. This high abort-rate behavior is caused by the large number of worker threads competing to read and modify a small set of records (cf. Figure 2.6). Unlike QueCC, any non-deterministic scheduling and concurrency control protocols will be a subject to significant and amplified abort rates when the number of conflicting operations by competing threads increases.

Effect of Write Operation Percentage Another factor that contributes to contention is the percentage of write operations. With read-only workloads, concurrency control protocols exhibit limited contention even if the data access is skewed. However, as the number



(a) Throughput

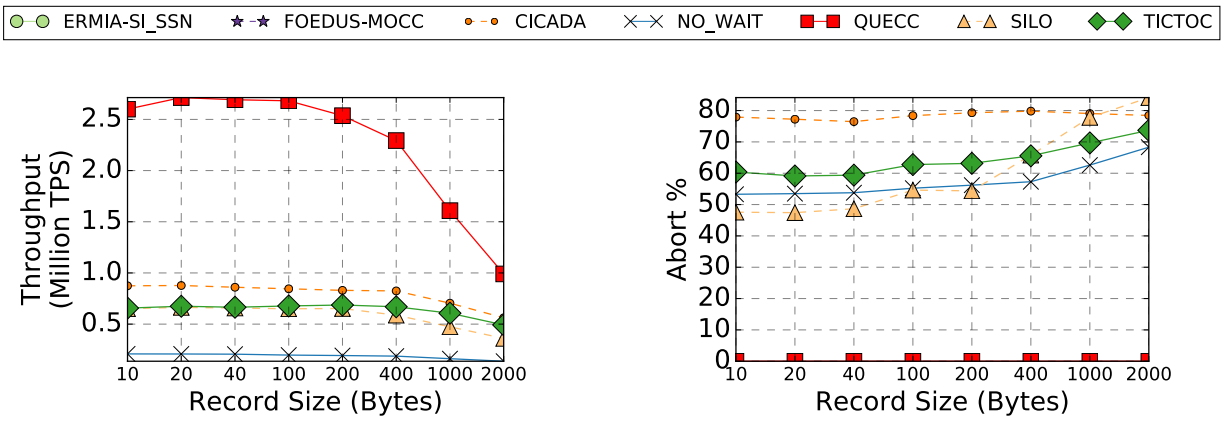
(b) Abort percentage

Figure 2.7. Results for varying the percentage of write operations in each transaction. High contention, $\theta = 0.99$

of conflicting write operations on records increases, the contention naturally increases, e.g., exclusive locks need to be acquired for NO-WAIT, more failed validations for SILO and Cicada, and in general, any approach relying on the optimistic assumption that conflicts are rare will suffer. Since QueCC does not perform any concurrency control during execution, no contention arise from the write operations.

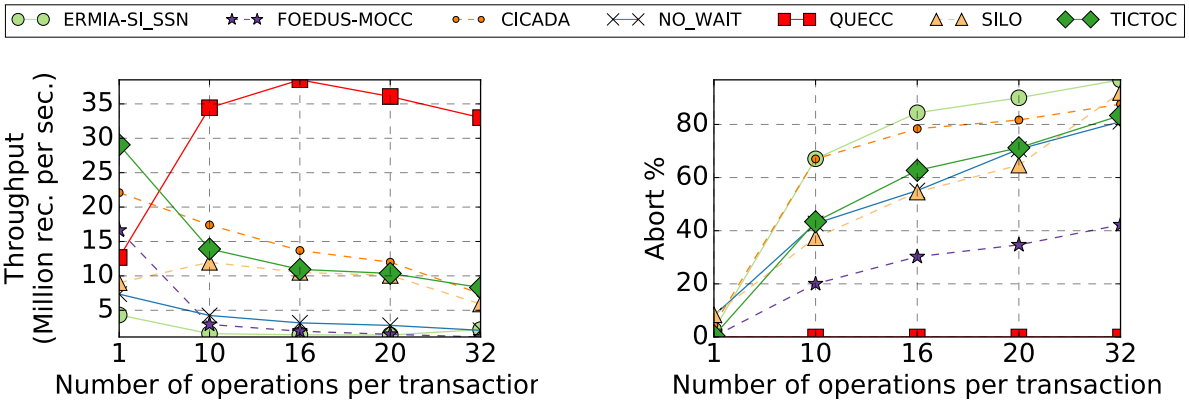
In addition to increased contention, write operations translates into increased size of undo logging for recovery. This is an added cost for any in-place update approach and QueCC is no exception. As we increase the write percentage, more records are backed up in the undo-buffers log and, thus, negatively impacts the overall system throughput. Of course, using a multi-version storage model (e.g., [30]) and avoiding in-place updates, the undo-buffer overhead can be mitigated. Nevertheless, QueCC significantly outperforms other concurrency control protocols by up to 4.5× under write-intensive workloads, i.e., once the write percentage exceeds 50%.

Effect of Record Sizes Having larger record sizes may also negatively affect the performance of logging component as shown in Figure 2.8. Since the undo log maintains a copy of every modified record, the logging throughput suffers when large records are used.



(a) Throughput (b) Abort Percentage

Figure 2.8. Results for varying the size of records under high contention, $\theta = 0.99$.

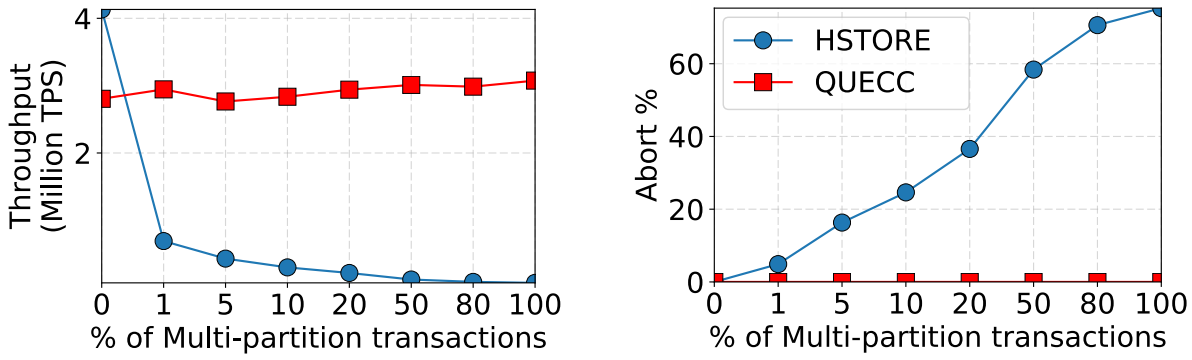


(a) Throughput (b) Abort percentage

Figure 2.9. Results for varying the number of operations in each transaction. High contention, $\theta = 0.99$

One approach to handle the logging is to exploit the notion of "abortabity" of the transaction last updated the record, and re-purpose the key principle of EWV[26].[4] Even under logging pressure that begins to become one of the dominant factor when the records size reaches $2KB$, QueCC continues to maintains its superiority and outperform Cicada by factor of 3× despite the contention regulation mechanism employed by Cicada.

[4]↑Similarly in QueCC, we check if all fragments of the last writer transaction has been executed successfully, if so, we avoid writing to the undo buffers, and we further avoid adding the commit dependency.



(a) Throughput

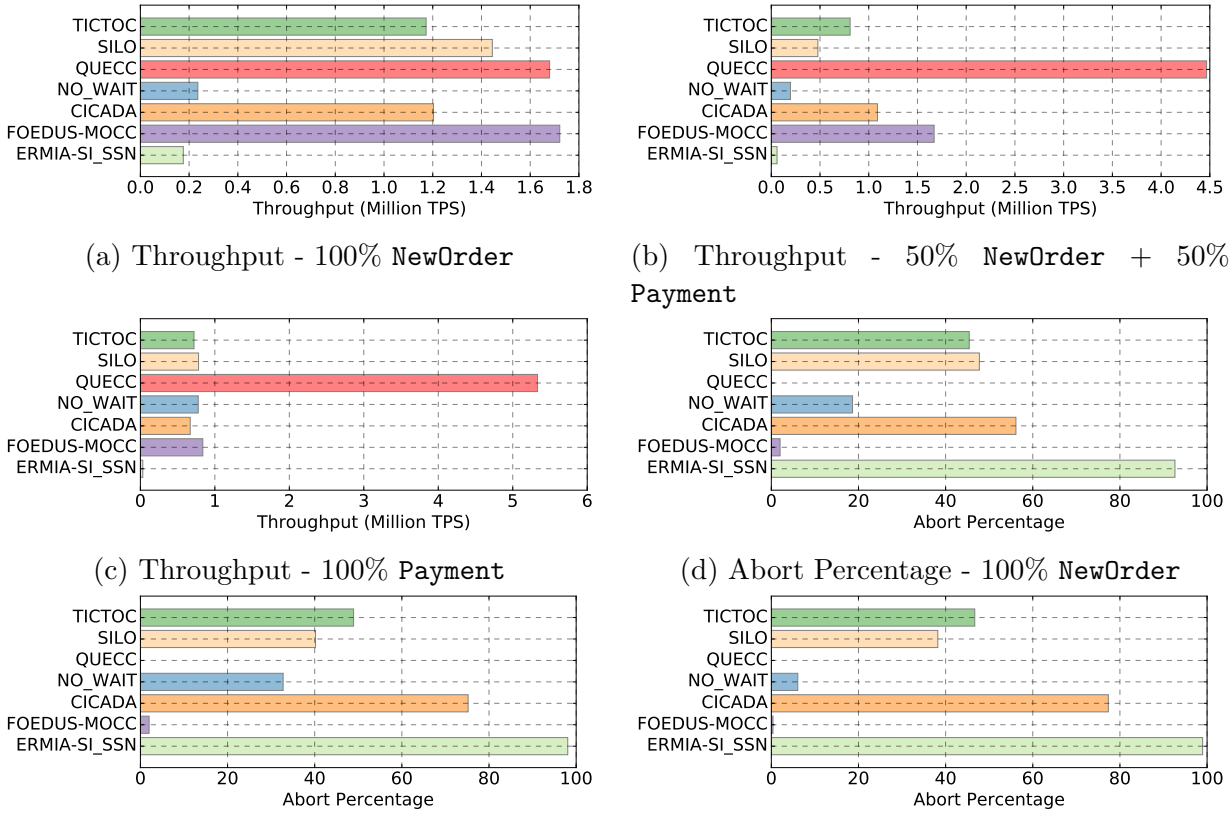
(b) Abort percentage

Figure 2.10. Results of multi-partition transactions with comparison to H-Store.

Effect of Transaction Size So far, each transaction contains a total of 16 operations. Now we evaluate the effect of varying the number of operations per transaction, essentially the depth of a transaction. Figure 2.9 shows the results of having $1, 10, 16, 20$, and 32 operations per transaction under high data skew. For these experiments, we report the throughput in terms of the number of operations completed or records accessed per second. For all concurrency control protocols, the throughput is lowest when there is only a single operation per transaction, which indicates that the work for ensuring transactional semantics is becoming the bottleneck.

More interestingly, when increasing the transaction depth, the probability of conflicting access is also increased; thereby, higher contention and higher abort rates. In contrast, under higher contention, QueCC continues to have zero percent abort rates. It further benefits from improved cache-locality and yields higher throughput because a smaller subset of records is handled by the same worker thread. QueCC further exploits intra-transaction parallelism and altogether improves up to 2.7× over the next best performing protocol (Cicada) when increasing the transaction depth.

Comparison to Partitioned Stores QueCC is not sensitive to multi-partition transactions despite its per-queue, single-threaded execution model, which is one of its key distinction. To establish QueCC's resilience to non-partition workloads, we devise an experiment in which we vary the degree of multi-partition transactions. Figure 2.10 illustrates that QueCC throughput virtually remains the same regardless of the percentage of multi-partition trans-



(a) Throughput - 100% `NewOrder`

(b) Throughput - 50% `NewOrder` + 50% `Payment`

(c) Throughput - 100% `Payment`

(d) Abort Percentage - 100% `NewOrder`

(e) Abort Percentage - 50% `NewOrder` + 50% `Payment`

(f) Abort Percentage - 100% `Payment`

Figure 2.11. Results for 32 worker threads for TPC-C benchmark. Number of warehouses = 1.

actions. We observed that QueCC improves over H-Store by factor 4.26× even when there is only 1% multi-partition transactions in the workload. Remarkably, with 100% multi-partition transactions, QueCC improves on H-Store by *two orders of magnitude*. H-Store is limited to a thread-to-transaction assignment and resolves conflicting access at the partition level. For multi-partition transactions, H-Store is forced to lock each partition accessed by a transaction prior to starting its execution. If the partition-level locks cannot be acquired, the transaction is aborted and restarted after some randomized delay. The H-Store coarse-grained partition locks offer an elegant model when assuming partition-able workload, but it noticeably limits concurrency when this assumption no longer holds.

2.5.4 TPC-C Experiments

In this section, we study QueCC using the industry standard TPC-C. Our experiments in this section focus on throughput and abort percentage under high contention with three different workload mixes.

From a data access skew point of view, the TPC-C benchmark is inherently skewed towards warehouse records because both `Payment` and `NewOrder` transactions access the warehouse table. The scale factor for TPC-C is the number of warehouses, but it also determines the data access skew. As we increase the number of warehouses, we get less data access skew (assuming a fixed number of transactions in the generated workload). Therefore, to induce high contention in TPC-C, we limit the number of warehouses to 1 in the workload and use all the 32 cores for processing the workload.

Figure 2.11 captures the throughput and the abort percentage. With a workload mix of 100% `Payment` transactions, Figure 2.11c, QueCC performs 6.34× better than the other approaches. With the a 50% `Payment` transaction mix, QueCC improves by nearly 2.7× over FOEDUS with MOCC. Despite the skewness towards the single warehouse record (where every transaction in the workload would accesses it), QueCC can process fragments accessing other tables in parallel because it distributes them among multiple queues, and assign those queues to different threads. In addition, QueCC performs no spurious aborts which contributes its high performance.

2.6 Related Work

There have been extensive research on concurrency control approaches, and there many excellent publications dedicated to this topic (e.g., [41]–[44]). However, research interest in concurrency control in the past decade has been revived due to emerging hardware trends, e.g., mutli-core and large main-memory machines. We will cover key approaches in this section.

Novel Transaction Processing Architectures Arguably one of the first papers that started to question the status quo for concurrency mechanism was H-Store [14]. H-Store imagined a simple model, where the workload always tends to be partitionable and advocated

single-threaded execution in each partition; thereby, drop the need for any coordination mechanism within a single partition. Of course, as expected its performance degrades when transactions span multiple partitions.

Unlike H-Store, QueCC through a deterministic, priority-based planning and execution model that not only eliminates the need for concurrency mechanism, but also it is not limited to partitionable workloads and can swiftly readjust and reassign thread-to-queue assignment or merge/spit queues during the planning and/or execution, where queue is essentially an ordered set of operations over a fine-grained partition that is created dynamically.

Unlike the classical execution model, in which each transaction is assigned to a single thread, DORA [11] proposed a novel reformulation of transactions processing as a loosely-coupled publish/subscribe paradigm, decomposes each transaction through a set of rendezvous points, and relies on message passing for thread communications. DORA assigns a thread to a set of records based on how the primary key index is traversed, often a b-tree index, where essentially the tree divided into a set of contiguous disjoint ranges, and each range is assigned to a thread. The goal of DORA is to improve cache efficiency using thread-to-data assignment as opposed to thread-to-transaction assignment. However, DORA continues to rely on classical concurrency controls to coordinate data access while QueCC is fundamentally different by completely eliminating the need for any concurrency control through deterministic planning and execution for a batch of transactions. Notably, QueCC's thread-to-queue assignment also substantially improve cache locality.

Concurrency Control Protocols The well-understood pessimistic two-phase locking schemes for transactional concurrency control on single-node systems are shown to have scalability problems with large numbers of cores[7]. Therefore, several research proposals focused on the optimistic concurrency control (OCC) approach (e.g., [8], [30], [35], [45]–[47]), which is originally proposed by [48]. Tu et al.'s SILO [8] is a scalable variant of optimistic concurrency control that avoids many bottlenecks of the centralized techniques by an efficient implementation of the validation phase. TicToc [35] improves concurrency by using a data-driven timestamp management protocol. Both BCC [46] and MOCC [45] are designed to minimize the cost of false aborts. All of these CC protocols suffer from non-deterministic aborts, which results in wasting computing resources and reducing the overall system's throughput. On

the other hand, QueCC does not have such limitation because it deterministically processes transactions, which eliminates non-deterministic aborts.

Larson et al. [29] revisited concurrency control for in-memory stores and proposed a multi-version, optimistic concurrency control with speculative reads. Sadoghi et al. [30], [47] introduced a two-version concurrency control that allows the coexistence of both pessimistic and optimistic concurrency protocols, all centered around a novel indirection layer that serves as a gateway to find the latest version of the record and a lightweight coordination mechanism to implement block and non-blocking concurrency mechanism. Cicada by Lim et al. mitigates the costs associated with multi-versioning and contention by carefully examining various layers of the system [9]. QueCC is in sharp contrast with these research efforts, QueCC focuses on eliminates the concurrency control overhead as opposed to improving it.

ORTHRUS by Ren et al. [17] uses dedicated threads for pessimistic concurrency control and message passing communication between threads. Transaction execution threads delegate their locking functionality to dedicated concurrency control threads. In contrast to ORTHRUS, QueCC plans a batch of transactions in the first phase and execute them in the second phase using coordination-free mechanism. LADS by Yao et al. [10] builds dependency graphs for a batch of transactions that dictates execution orders. Faleiro et al. [26] propose PWV which is based on the "early write visibility" technique that exploits the ability to determine the commit decision of a transaction before it completes all of its operations. In terms of execution, both LADS and PWV process transactions *explicitly* by relying on dependency graphs. On the other hand, QueCC does satisfy transaction dependencies but its execution model is organized in term of prioritized queues. In QueCC, not only do we drop the partitionability assumption, but we also eliminate any graph-driven coordination by introducing a novel deterministic, priority-based queuing execution. Notably, the idea of "early write visibility" can be exploited by QueCC to further reduce chances of cascading aborts.

The ability to parallelize transaction processing is limited by various dependencies that may exist among transactions fragments. IC3 [28] is a recent proposal for a concurrency control optimized for multi-core in-memory stores. IC3 decomposes transactions into pieces

through static analysis, and constrain the parallel execution of pieces at run-time to ensure serializable.

Unlike IC3, QueCC achieves transaction-level parallelism by using two deterministic phases of planning and execution and without relying on conflict graphs explicitly.

Deterministic Transaction Processing All the aforementioned single-version transaction processing schemes interleave transaction operations non-deterministically, which leads to fundamentally unnecessary aborts and transaction restarts. Deterministic transaction processing, e.g.,[12], [16]) on the other hand, eliminates this class of non-deterministic aborts and allow only logic-induced aborts (i.e., explicit aborts by the transaction's logic). Calvin[12] is designed for distributed environments and uses determinism eliminate the cost of two-phase-commit protocol when processing distributed transactions and does not address multi-core optimizations in the individual nodes. Gargamel [49] pre-serilaize possibly conflicting transactions using a dedicated load-balancing node in distributed environments. It uses a classifier based on static analysis to determine conflicting transactions. Unlike Gargamel, QueCC is centered around the notion of priority, and is designed for multi-core hardware.

BOHM [16] started re-thinking multi-version concurrency control for deterministic multi-core in-memory stores. In particular, BOHM process batches of transactions in three sequential phases (1) a single-threaded sequencing phase to determine the global order of transactions, (2) a parallel multi-version concurrency control phase to determine the version conflicts, and (3) a parallel execution phase based on transaction dependencies, which optionally performs garbage collection for unneeded record versions. In sharp contrast, QueCC process batches of transactions in only two deterministic phases, and it has a parallel priority-based queue-oriented planning and execution phases that do not suffer from additional costs such as garbage collection costs.

2.7 Conclusion

In this chapter, we presented QueCC, a *queue-oriented, control-free concurrency architecture* for high-performance, in-memory data stores on emerging multi-sockets, many-core, shared-memory architectures. QueCC exhibits minimal contention among concurrent threads

by eliminating the overhead of concurrency control from the critical path of the transaction. QueCC operates on batches of transactions in two deterministic phases of priority-based planning followed by control-free execution. Instead of the traditional thread-to-transaction assignment, QueCC uses a novel thread-to-queue assignment to dynamically parallelize transaction execution and eliminate bottlenecks under high contention workloads. We extensively evaluate QueCC with two popular benchmarks. Our results show that QueCC can process almost 40 Million YCSB operation per second and over 5 Million TPC-C transactions per second. Compared to other concurrency control approaches, QueCC achieves up to 4.5× higher throughput for YCSB workloads, and 6.3× higher throughput for TPC-C workloads.

3. QUEUE-ORIENTED DISTRIBUTED TRANSACTION PROCESSING

An earlier version of this chapter appeared in [50]

3.1 Introduction

Distributed transaction processing is challenging due to the inherent overheads of costly commit protocols like *2-Phase-Commit* (2PC) [51]. Even for use cases such as in-memory databases and stored-procedure-based transactions, 2PC is either used (e.g., [14], [52], [53]) or avoided by eliminating the processing of multi-partitioned transactions (e.g., [54], [55]). Note that 2PC by itself does not ensure serializable transaction processing, and it requires a distributed concurrency control protocol to guarantee serializability. Traditional concurrency control protocols may abort active distributed transactions non-deterministically to ensure serializable transaction processing. When such abort decisions are coupled with 2PC, the cost of the distributed transaction processing is further increased because of the overhead of rollbacks and restarts.

Deterministic databases [12], [56] reduce the cost of committing distributed transactions by imposing a single order on executing a batch of transactions prior to actual execution. By ensuring the same pre-execution ordering, deterministic database systems eliminate the need to abort transactions for violating serializability guarantees (in optimistic concurrency control), avoiding deadlocks (in pessimistic concurrency control), or node crash failures.

Unfortunately, the state-of-the-art designs of distributed deterministic databases suffer from other inefficiencies. We identify three of these inefficiencies that limit their performance and scalability. First, they rely on single-threaded pre-execution sequencing and scheduling mechanisms which cannot exploit multi-core computing architectures and limit vertical throughput scalability [13]. Second, they mostly support a conservative (non-speculative) form of transaction execution. One exception is the work by Jones et al. [52], which performs speculative-execution only for multi-partition transactions but limits the concurrency for single-partition transactions (a property inherited from H-Store's design). Third, they follow a thread-to-transaction assignment which limits intra-transaction parallelism [2], [11].

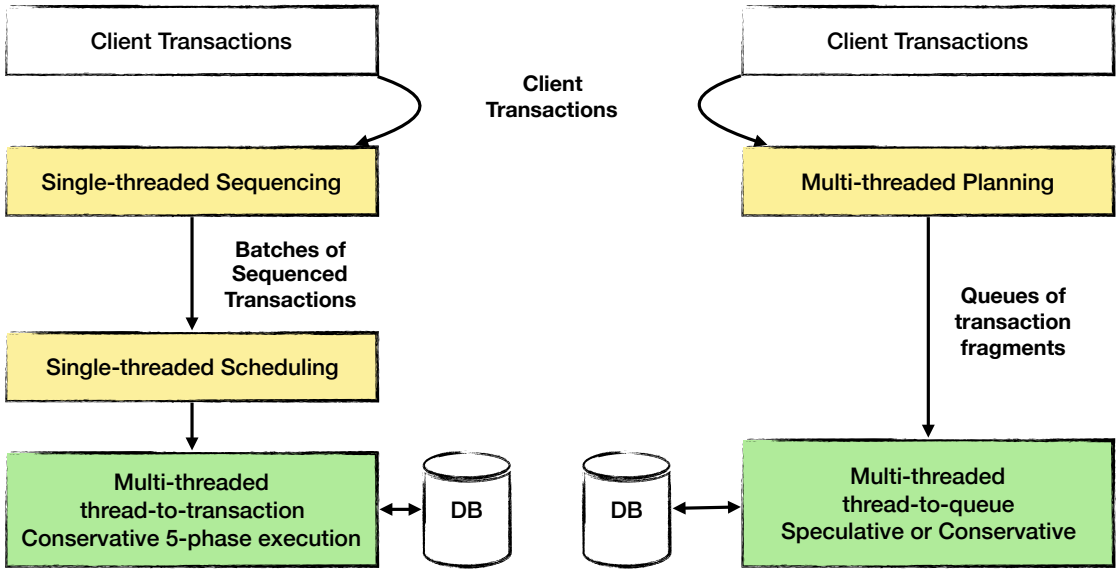


Figure 3.1. Overview of transaction processing in Calvin (left) and Q-Store (right)

In this chapter, we propose a novel transaction processing paradigm of *queuing-oriented processing*, and describe Q-Store. Q-Store is built on the principles of queue-oriented paradigm, which provides a unified abstraction for processing distributed transactions deterministically and does not suffer from the inefficiencies such as lower utilization of cores. Furthermore, it admits multiple execution paradigms (i.e., *speculative* or *conservative*) and multiple isolation levels (i.e., serializable isolation or read-committed isolation) seamlessly, unlike existing proposals of the deterministic database. It is important to note that several existing non-deterministic database systems already support multiple forms of isolation levels (e.g., [30], [57]–[60]).

Our *queue-oriented* transaction processing paradigm can efficiently utilize the parallelism available with commodity multi-core machines by maximizing the number of threads doing useful work. Q-Store processes batches of transactions in two multi-threaded yet deterministic phases of *planning and execution*, as shown in the right side of Figure 3.1. Each phase utilizes all available computing resources efficiently, which improves the system's throughput significantly. The planning phase is carried out by multiple *planning-threads*, delivering maximum CPU utilization. Planning-threads generate queues of transaction operations that require minimal coordination among *execution-threads*. These queues are executed by

execution-threads that are assigned to different cores to maximize cache efficiency. Each execution-thread is assigned one or more queues for execution. In other words, these queues constitute a schedule for executing transnational operations of a batch of transactions. Any coordination among execution-threads is performed via efficient and distributed lock-free data structures. In particular, we make the following contribution, in this chapter.

- We propose a novel queue-oriented transaction processing paradigm that facilitates distributed transaction processing and unifies local and remote transaction processing in a single paradigm based on pre-determined priorities of queues. Our proposed paradigm supports multiple execution paradigms and multiple isolation levels and leads to implementation of efficient transaction processing protocol (Section 3.3).

- We present a formalization of our proposed paradigm and prove that it produces serializable histories to guarantee serializable isolation. We also formally show how our paradigm can support read-committed isolation seamlessly (Section 3.4).

- We design and build Q-Store, which is a distributed transaction processing system that relies on the principles of our proposed queue-oriented paradigm (Section 3.5).

- We present the results of an extensive evaluation of Q-Store. In our evaluation, we compare Q-Store against non-deterministic and deterministic transaction processing protocols using workloads from standard macro-benchmarks such as YCSB and TPC-C. We perform our evaluation using a single code-base, which allows us to conduct an apple-to-apple comparison against 5 transaction processing protocols. Our experiments demonstrate that Q-Store out-performs state-of-the-art deterministic distributed transaction processing protocols by up to 22.1×. Against non-deterministic distributed transaction processing protocols, Q-Store achieves up to two orders of magnitude better throughput (Section 3.6).

3.2 Background

In this section, we give an overview of Calvin [12] as a representative for deterministic databases. As far as we know, Calvin is regarded as the state-of-the-art distributed determin-

istic transaction processing protocol, and has been commercialized [56]. Other deterministic transaction processing protocols are either designed for non-distributed environments (e.g., [2], [15], [16]) or a variation that improves parts of Calvin's protocol while re-using the remaining parts as-is (e.g., [61]). These proposals are covered in Section 3.7 in more details. We also briefly describe the transaction model used by Q-Store which adopts the same transaction model used by [2], which is in sharp contrast from Calvin's transaction model.

3.2.1 Transaction Processing in Calvin

This section gives a brief description of how Calvin works based on [12]. The basic processing flow requires 3 phases: a sequencing phase, a scheduling phase, and an execution phase with 5 sub-phases. Figure 3.1 (left), illustrate these phases.

Each node, in Calvin, runs a single sequencer thread, a single scheduler thread, and one or more worker threads. The sequencer thread forms batches of sequenced transactions. It uses a time-based demarcation of batches. Batches formed by different nodes are processed by scheduler threads in strict round-robin fashion. Scheduler threads use deterministic locking to schedule transactions that require the full knowledge of the read/write sets of transactions, which is similar to *Conservative 2PL* [62]. Unlike Conservative 2PL, Calvin ensures that conflicting transactions are deterministically processed according to their sequence number in the sequencing batch. For example, let t_a and t_b denote two conflicting transactions (i.e., cannot be scheduled to execute concurrently), and $seq(t)$ denote the sequence number of transaction t as determined by the sequencer thread. If $seq(t_a) < seq(t_b)$, then Calvin ensures that t_a is scheduled before t_b. Once locks on all the records are acquired by the scheduler thread, the transaction is ready for execution, and it is given to a worker thread for execution.

As Calvin is a distributed database system, each worker thread executes an assigned transaction in the following phases:

Phase 1 - Read/write set analysis: This phase is used to determine the set of nodes that are participating in the transaction. For this set, nodes that are executing at least one write operation are marked as active participants.

Phase 2 - Perform local read operations: This phase is performed by all participants if records are available locally.

Phase 3 - Serve remote read operations: Multicast records to active participants. This phase is the last phase performed by non-active participants. At this point, they can declare the transaction as completed and move to the next transaction.

Phase 4 - Collect remote read operations: This phase is performed by active participants only, and they need to wait for remote records before moving to the next phase. Hence, worker threads can postpone the active transaction (while waiting) and resume another transaction that is ready for execution.

Phase 5 - Execute transaction logic and perform local write operations: This phase is also performed only by active participants.

Discussion. The original Calvin paper by Thomson et al. [12] does not clearly describe how a transaction is committed (or aborted). However, by looking into the code-base of one of the implementations of Calvin from [13], which we ported to our test-bed, we discovered that the basic idea goes as follows.

The sequencer determines the participant nodes of every sequenced transaction by performing **Phase 1** from above. When a participant node completes its work on a transaction, it sends a one-way acknowledgment (ACK) message to the sequencer of the transaction. When the sequencer collects all ACK messages from all participants, it commits the transaction and sends a response message to the client of the transaction. Worker threads (that execute transactions) can re-use read/write set analysis performed by the sequencer thread to avoid needless computation.

3.2.2 Q-Store's Transaction Model

We adopt the same transaction model used by [2]. In this model, a transaction is broken into fragments. A fragment can perform multiple operations on the same record, such as read, modify, and write operations. A fragment can cause the transaction to abort, and in this case, we refer to such fragments as abortable fragments.

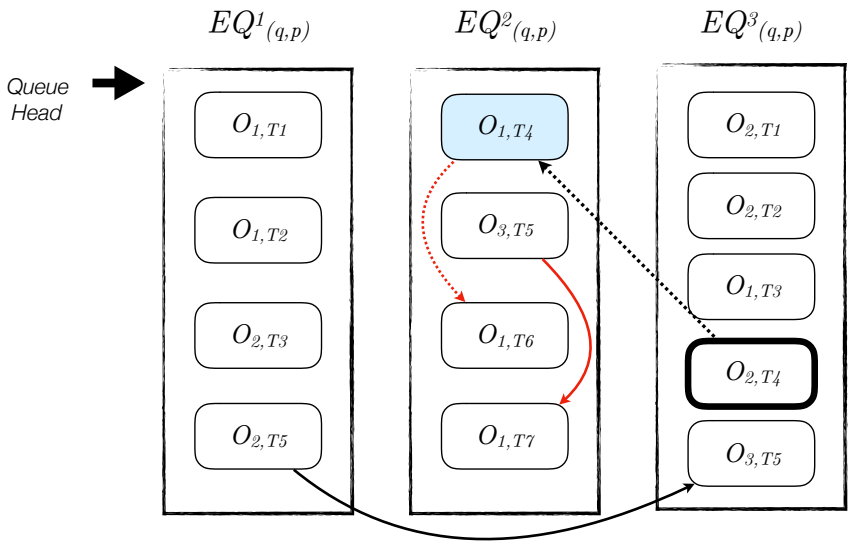


Figure 3.2. An example illustrating transaction dependencies in Q-Store. Execution-queues (EQs) are planned by planning-thread $PT_{(q,p)}$

Furthermore, there are can be dependencies among fragments. In Figure 3.2, we illustrate these dependencies. There are 7 planned transactions in 3 execution-queues. Fragments are denoted as $O_{\mathrm{i},T_\mathrm{j}}$ where i denotes the fragment index in transaction T_j. We describe the notations in detail in Section 3.4.

Data dependencies exist when an operation in a fragment requires a value that is read by another fragment of the same transaction (solid black arrow between O_{2,T_5} and O_{3,T_5}).

Conflict dependencies exist between fragments from different transactions that access the same record, and the dependee fragment performs a write operation (solid red arrow between O_{3,T_5} and O_{1,T_7}).

Two kinds of commit dependencies exist between fragments. The first kind is concerned with fragments of the same transaction. In this case, a commit dependency exists between two fragments of the same transaction if the dependee is an *abortable* fragment (dotted black arrow between O_{2,T_4} and O_{1,T_4}). In this example, O_{2,T_4} is an abortable fragment. The second kind of commit dependencies, which we refer to as *speculation dependencies*, exist between fragments of different transactions. Tracking them is required when using the speculative execution paradigm. A speculation dependency exists between the two fragments when the

dependent fragment reads speculatively uncommitted data written by the dependee fragment (dotted red arrow between O_{1,T_4} and O_{1,T_6}).

Discussion. It is worth noting that speculation dependencies are a realization of conflict dependencies. Tracking speculation dependencies is needed to ensure correct transaction execution with speculative execution. Note that, in Q-Store, conflict dependencies are not explicitly tracked during planning. It is possible to capture these during planning, but that would introduce additional overhead to the planning phase, which is undesirable.

3.3 Transaction Processing in Q-Store

In this section, we describe the novel and unique features of Q-Store. As far as we know, Q-Store is the first distributed deterministic transaction processing system to provide following features.

Efficient two-phase distributed processing model. In Figure 3.1, we show the critical differences between Calvin's processing model and Q-Store' processing model. On the left side (Calvin) of Figure 3.1, the total number of phases required to process a batch of transactions is 3 with the execution phase requiring 5 sub-phases. Note that the sequencing and the scheduling phases in Calvin are single-threaded. On the right side, Q-Store processes a batch of transactions in two multi-threaded phases of planning and execution. The execution phase does not include any sub-phases. Q-Store reduces the number of phases compared to Calvin (See Figure 3.1). Furthermore, Q-Store uses all available cores efficiently. All available threads work on the planning of a batch, then all of them work on execution.

Multi-paradigm execution. The design of Q-Store admits multiple execution paradigm. The processing of a batch of transactions can be speculative or conservative. Transaction isolation can be serializable or read-committed.

Queue-oriented processing. The planning phase in Q-Store abstracts the logical semantics of a transaction into prioritized queues of transaction fragments. Queues provide ordering for conflict fragments that seamlessly resolve conflict dependencies among fragments of different transactions. Therefore, threads during execution only deal with the other de-

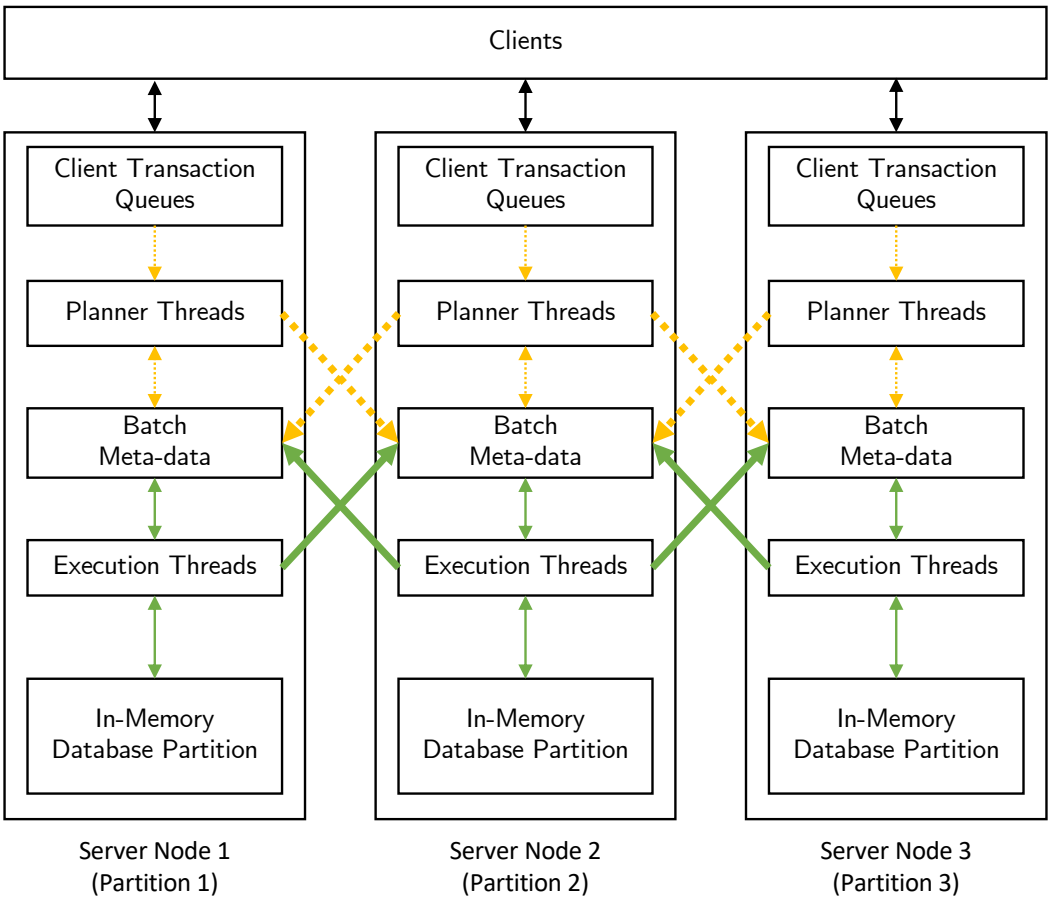


Figure 3.3. System Architecture

pendencies. Furthermore, queues can be implemented efficiently to ensure efficient execution and communication.

3.3.1 Queue-oriented Architecture

In Figure 3.3, we illustrate an example architecture of Q-Store, which consists of three server nodes. A client may send transactions that require access to multiple partitions, which we call multi-partition transactions. A client selects one of the server nodes for a given transaction and sends the transaction to the selected server. The role of the selected server is to coordinate the execution of the received transaction. Note that a server can be selected during the client session establishment, which allows mechanisms for load-balancing. Mech-

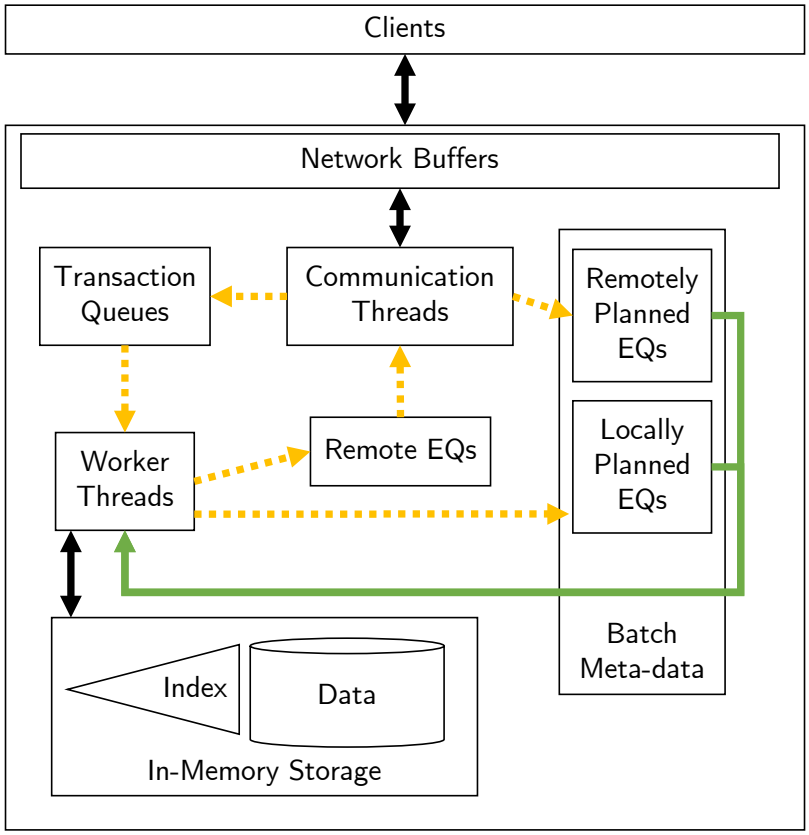


Figure 3.4. Server Node Architecture

anisms for load-balancing include client-side libraries and middle-ware-based mechanisms. These details are beyond the scope of this chapter.

Also in Figure 3.3, each node maintains a set of local *client transaction queues*. There is one client transaction queue per planner-thread to avoid contention. Planner-threads create fragments from transactions and capture dependencies, and create queues of fragments for each execution thread. Each planner-thread also updates the *Batch Meta-data* distributed data structure. The Batch Meta-data stores information about fragment dependencies, and execution-queues progress status. It is a globally shared lock-free distributed data structure that is used to facilitate minimal coordination among execution threads. In Figure 3.3, yellow arrows depict communication patterns during the planning-phase while green arrows depict communication patterns during the execution-phase.

Zooming into a single node, Figure 3.4, we illustrate the major components of a server node. Similar to Figure 3.3, yellow and green arrows, depict communication during planning-phase and execution-phase, respectively.

Each server employs a set of threads to complete various tasks. We can broadly categorize threads into two sets: (i) *communication threads*, and (ii) *worker threads*. Q-Store employs communication threads to handle message transmission and reception among the servers and clients. They are also responsible for handling messages between server components and network buffers. They store client transactions in transaction queues, send and receive remote execution-queues (EQs), and apply updates to the batch meta-data.

Each worker thread may participate in either one or two phases: *planning* and *execution* (e.g., we can have dedicated worker threads for each phase). Hence depending on the phase, we refer to these worker threads as either *planning-threads* or *execution-threads*. We use $\mathbb{P}$ to represent the set of planning-threads and $\mathbb{E}$ to represent the set of execution-threads. The planning-threads take a set of transactions and generate *plans* to execute these transactions. The execution-threads execute transactions according to these plans.

3.3.2 Priorities in Q-Store

In Q-Store, we use the notion of priorities to impose order at various levels granularity. The concept of priority captures the ordering of queues and transaction fragments elegantly. Execution-threads need to respect these priorities to ensure the correct ordering of conflicting transactions. We have three different levels of granularity from the perspective of execution.

We formalize the notion of priorities by representing our distributed system as a set $\mathbb{S}$, which is the set of server nodes. We assign each server S_q a priority q, that is,

$$\mathbb{S} := \{S_1,\ S_2, ..., S_q\},\ \text{where } q \geq 1$$

Q-Store requires each server to associate a priority p with each of its planning-threads. Note that the planning-thread priority p differs from the server priority q. As each planning-

thread also inherits the priority of its server, so each planning-thread has two associated priorities. Hence, we use the $P_{q,p}$ representation for a planning thread with priority p.

$$\mathbb{P}_q := \{P_{q,1},\ P_{q,2}, ...,\ P_{q,p}\},\ \text{where } q, p \geq 1 \tag{3.1}$$

Planning-threads create execution-queues for transactions and tag them with their priorities. The execution-queues created by a planning-thread constitute schedules of transaction fragments of the set of transactions processed by the planning-thread. Execution-threads execute fragments according to planned schedules while respecting the priorities of execution-queues in addition to checking and resolving dependencies among fragments.

3.3.3 Logging and Recovery

Q-Store like other deterministic transaction processing systems (e.g., [12], [14]) assumes a deterministic stored procedure based transaction model [63]. Within this model, all inputs of a transaction are available before this transaction can start execution. Therefore, the input of a batch of transactions is logged before they are delivered to execution-threads. Periodic check-pointing of the database state is used to reduce the time required for recovery in case of a failure. In this chapter, we mainly focus on transaction execution as we can rely on the same techniques for logging and recovery as [12], [14].

3.4 Formalizing Q-Store

We now formalize the planning and execution phases of Q-Store. Later in this section, we also prove that Q-Store transaction processing protocol produces serializable histories.

3.4.1 Planning Transactions

As stated earlier in the previous section, the set of planning-threads $\mathbb{P}_q$ at a server inherit its priority q, and each planning-thread $P_{q,p}$ in the set $\mathbb{P}_q$ has another priority p to prioritize planning-threads of the same server. In general, planning-threads may use any mechanism to create execution-queues as long as they ensure that conflicting operations are placed in the

same queue. For example, a *range*-based partitioning of the *record-identifiers* can be used, which ensures that operations accessing the same record are placed in the same execution queue. However, a placement strategy that minimizes the dependencies among execution-queues can yield better performance. More sophisticated approaches based on some cost model are also possible as long as the planning times are minimized and do not introduce significant overhead to the processing latency. The study of such strategies is out of the scope of this chapter.

We denote the set of these execution-queues as $\mathbb{Q}_{q,p}$ and individual execution-queue as $Q^{\mathrm{i}}_{q,p}$.

$$\mathbb{Q}_{q,p} := \{Q^1_{q,p},\ Q^2_{q,p}, ..., Q^{\mathrm{i}}_{q,p}\},\ \text{where i} \geq 1 \tag{3.2}$$

In Q-Store, each planning-thread processes a batch of transactions and places its fragments in its respective execution-queues. Hence, each i^{th} execution-queue $Q^{\mathrm{i}}_{q,p}$ contains a set of operations that access the records belonging to that sub-partition, which implies that fragments from two execution-queues created by the same planning-thread have operations, access records in different sub-partitions, and any conflicting fragments (i.e., access the same records) are placed in the same execution-queue.

Q-Store's planning-threads try to balance the load and create execution-queues equal to the number of execution-threads in the system. Such planning is done to keep execution-threads from being idle. More formally:

$$\forall\ P_{q,p}, \in \mathbb{P}_q,\ |\mathbb{Q}_{q,p}| \geq |\mathbb{E}| \tag{3.3}$$

However, it is undesirable in practice to have the number of execution-queues much larger than the number of execution-threads because it can lead to performance degradation due to low-level issues (e.g., cache-locality).

Each transaction can perform multiple operations. These operations can be grouped into fragments if they are accessing the same record. Otherwise, a fragment has a single operation. We denote the set of fragments in a transaction T as $\mathcal{O}_T$. For presentation

simplicity, let us assume that each fragment $O_{k,T}$ in set $\mathcal{O}_T$ can be either a read (R) or a write (W).

$$\mathcal{O}_T := \{O_{1,T},\ O_{2,T}, ..., O_{k,T}\}$$

A planning-thread may distribute the fragments of a given transaction across multiple execution-queues. Q-Store needs to impose an order to the transactions that are being planned. This order can be as simple as the order imposed by the *client-transaction-queue*. A transaction and its fragments inherit the priorities of its planning-thread.

Hence, we can identify the order of a transaction T using a triple (i, p, q), where q is the priority of the server, p is the priority of the planning thread, and i can be the order imposed by the client-transaction-queue.

$$\forall\ \mathrm{i}, \mathrm{j},\ \mathrm{i} < \mathrm{j} \rightarrow T_{(\mathrm{j},p,q)} \xrightarrow{\text{follows}} T_{(\mathrm{i},p,q)} \tag{3.4}$$

Equation 3.4 shows that as $T_{(\mathrm{i},p,q)}$ has a smaller identifier (i) than $T_{(\mathrm{j},p,q)}$, so it must have been placed in the client-transaction-queue before $T_{(\mathrm{j},p,q)}$.

Since transactions may have operations accessing remote partitions, planning-threads similarly create remote execution-queues to be executed at remote nodes. Note that our notation of an execution-queue $Q^{\mathrm{i}}_{q,p}$ identifies the priority q of a remote server, which guides the execution phase. Therefore, queue execution at remote nodes is also deterministic.

When the planning-threads have collectively processed a set of transactions, they mark the resulting batch of execution-queues (local and remote) as ready for execution and deliver them to (local and remote) execution threads.

3.4.2 Speculatively Executing Transactions

The execution phase is performed by a set of execution-threads. Each server consists of a set of execution-threads $\mathbb{E}$.

$$\mathbb{E} := \{E_1, E_2, ..., E_{\mathrm{j}}\},\ \text{where } \mathrm{j} \geq 1$$

We require all the execution-threads to adhere to the following condition strictly:

`Condition`: *For each record, operations belonging to higher priority execution-queues must always be executed before executing any lower priority operations.*

$$\begin{aligned} &\forall\ Q_m \in \mathbb{Q}_{q,p},\ \forall\ Q_n \in \mathbb{Q}_{s,t},\ \forall\ O_{\mathrm{i}} \in Q_m,\ \forall\ O_{\mathrm{j}} \in Q_n \\ &\mid (q > s)\ \vee\ ((q = s)\ \wedge\ (p > t)) \rightarrow O_{\mathrm{j}} \xrightarrow{\text{follows}} O_{\mathrm{i}} \end{aligned} \tag{3.5}$$

This condition ensures that the order of executed operations follows a single order within and across servers. In other words, Q-Store requires execution-threads $\mathbb{E}$ to process the operations from those execution-queues, which have the highest priority among all the servers and planning-threads. However, Q-Store does allow the execution-queues produced by a single planner thread to be executed in parallel because they have the same priority.

Execution-threads process fragments from the execution-queues speculatively such that fragments are allowed to read uncommitted data (speculating that it would commit at a later time). Q-Store tracks these speculative actions and captures corresponding speculation dependencies (Section 3.2.2).

When a violation of an integrity constraint causes a transaction to abort, other fragments of the same transaction that have updated records must rollback as well. The other fragments may have uncommitted updates that have been read by fragments belonging to other transactions. In this case, dependent fragments and their respective transactions must rollback, causing a cascade of aborts through the batch.

3.4.3 Conservatively Executing Transactions

Q-Store also seamlessly supports a conservative execution, which introduces stalls when processing queues, but has the advantage of avoiding cascading aborts. In Q-Store, a transaction is aborted when the transaction logic induces an abort (e.g., for violating an integrity constraint). By design, non-deterministic aborts (e.g., for ensuring deadlock-free execution) do not exist in Q-Store.

Looking back at our example illustrating transaction dependencies from Section 3.2.2, Fragment O_{1,T_4} depends on O_{2,T_4} which is abortable. In conservative execution, the execution-thread executing $EQ^2_{q,p}$ stalls until the dependency is resolved. The event of resolving the dependency indicates that O_{2,T_4} is not going to abort. Therefore, any records updated by fragment O_{1,T_4} are safe for any read operations by subsequent fragments in the execution-queue.

Fragments are marked by planning-threads to ensure that execution-threads know when to wait and stall the processing of an execution-queue. When execution-threads encounter a marked fragment, they stall waiting for its commit dependencies to be resolved. Execution-threads can work on other execution-queues if they need to stall due to unresolved commit dependencies. Therefore, we are still exploiting parallelism by allowing other fragments to execute. If an integrity constraint violation happens, then, only one transaction is aborted and rollbacked.

3.4.4 Serializability

We now prove the serializability guarantees of Q-Store's transaction processing model.

Theorem 3.4.1. Q-Store*'s distributed transaction processing is serializable.*

Proof. One principle of our queue-oriented paradigm is to treat local and remote execution-queues in the same way. Therefore, the fact that an execution-queue is remote or local is an orthogonal concept.

Let us assume that Q-Store produces a non-serializable history, which means that there exist 4 transaction fragments that are executed in an incorrect order. Let these fragments be as follows: $O_{\mathrm{i},T_n}, O_{\mathrm{j},T_n}, O_{k,T_m}$ and O_{l,T_m}. Here O_{i,T_n} conflicts with O_{k,T_m} and O_{j,T_n} conflicts with O_{l,T_m}. Further, let $n < m$ in the client transaction queue, which means that a planner plans T_n before T_m. A non-serializable history means that at one execution-queue O_{i,T_n} is executed before O_{k,T_m} while O_{k,T_m} is executed before O_{i,T_n} at another execution node. More formally,

$$\exists Q_a, Q_b \text{ s.t. } \{O_{\mathrm{i},T_n}, O_{k,T_m}\} \in Q_a \wedge \{O_{\mathrm{j},T_n}, O_{l,T_m}\} \in Q_b \tag{3.6}$$

Furthermore, the following constraint captures one possible non-serializable history for the transaction fragments.

$$O_{k,T_m} \xrightarrow{\text{follows}} O_{\text{i},T_n} \wedge O_{\text{j},T_n} \xrightarrow{\text{follows}} O_{l,T_m} \tag{3.7}$$

The other non-serializable history is captured by:

$$O_{\text{i},T_n} \xrightarrow{\text{follows}} O_{k,T_m} \wedge O_{l,T_m} \xrightarrow{\text{follows}} O_{\text{j},T_n} \tag{3.8}$$

Either Q_a or Q_b has fragments from T_m ordered before T_n, which contradicts the fact that the planner of Q_a and Q_b planned T_n before T_m.

□

3.4.5 Read-committed Isolation

Not only that, Q-Store supports multiple execution paradigms but also multiple isolation levels seamlessly using the queue-oriented paradigm. Supporting read-committed isolation requires planning-threads to produce an additional set of execution-queues $Q^{\text{j}}_{q,p}$ such that they only contain read-only transaction fragments as shown in Eq. 3.9. Read-only transaction fragments do not perform any write operations.

$$\begin{aligned} \forall\ P_{q,p}, \in \mathbb{P}_q, \mathbb{Q}_{q,p} := \{Q^1_{q,p},\ Q^2_{q,p}, ..., Q^{\text{i}}_{q,p}\} \\ \cup \{Q^1_{q,p},\ Q^2_{q,p}, ..., Q^{\text{j}}_{q,p}\},\ \text{where i} \geq 1, \text{j} \geq 1 \end{aligned} \tag{3.9}$$

Furthermore, Q-Store employs a copy-on-write technique that creates a private copy of the updated records. Using these two simple techniques, Q-Store can support read-committed isolation seamlessly.

3.4.6 Discussion

The performance of speculative execution is dependent on the workload. Two properties of the workload can degrade the performance of speculative execution. The activation of

logic-induced aborts which leads to the cascading aborts phenomena. The conservative execution solves this issue at the cost of more coordination among execution-threads.

For either of the execution paradigms, there is another workload property that impacts their performance negatively. The existence of a large number of data dependencies among fragments (see Section 3.2.2) in the planned workload limits the concurrency because it forces additional coordination among threads to resolve these data dependencies.

In Q-Store, mitigating the impact of data dependencies require more intelligent planning. Planning-threads can minimize the data dependencies among execution-queues. However, solving the minimization problem cannot introduce significant latency. Furthermore, because the database is partitioned, this can only work for local execution-queues. Planning-threads can intelligently move read-only fragments to a special set of execution-queues that allow resolving data-dependencies before executing dependent fragments. The implementation of these optimization remains as future work.

3.5 Implementation

We now present some key details for our implementation of Q-Store. In our implementation of Q-Store, we model various components of Q-Store as a set of producers and consumers. As stated in Section 3.3, Q-Store includes a set of communication-threads. These threads perform two tasks: (i) consuming messages from the network and storing them in respective queues, and (ii) consuming messages from the worker-threads and pushing those on to the network. The task of consuming messages from the network involves reconstructing the raw buffers into appropriate message types so that other threads can interpret them.

In Q-Store, we partition the database using a range-partitioning scheme. At each server, we allocate an equal number of worker threads that assume the roles of both the planning-threads and execution-threads but only one role at a time. This scheme simplifies both the planning and execution phases as computing the number of sub-partitions across the whole cluster requires no additional communication.

When an input thread receives a client transaction, it places the transaction into a client-transaction-queue associated with one of the planning-threads, in a round-robin fashion. We

allocate one client-transaction-queue for each planning-thread. This approach eliminates contention among the planning-threads to fetch the next transaction.

Q-Store employs a *count-based batch demarcation* mechanism which requires Planning-threads to create batches of transactions containing a specific number of transactions. However, *time-based implementations* for defining batches are also possible (e.g., a batch is created every 5 milliseconds).

Our Q-Store's implementation requires minimal low-level synchronization among all the threads in the system. Communication threads and worker threads utilize lock-free data structures to interact. For instance, if a worker thread is currently acting as a planning-thread, then as soon as it has processed the required number of transactions for the next batch and created its execution-queues, it starts acting as an execution-thread and checks for any available execution-queue to process. When it has executed all the required execution-queues, then it resumes the role of a planning-thread.

Batch Meta-data Q-Store requires execution-threads to process both the local execution-queues and remote execution-queues. This requirement implies there is a need to store locally generated execution-queues and incoming remote execution-queues. We employ a distributed lock-free data-structure, which we refer to as the *Batch meta-data* (illustrated in Figures 3.3 and 3.4), to store these execution-queues as well as any relevant meta-data needed to fulfill transactions dependencies. The implementation of dependencies uses a count to represent the number of dependencies to be resolved. When a dependency is resolved, we use atomic operations to decrement the dependency count. The communication-threads push the incoming remote execution-queues directly to the batch meta-data, which makes these queues available for execution. In this case, communication-threads are acting as virtual planning-threads. Execution-threads access this batch meta-data to fetch any available remote execution-queues. Moreover, the batch meta-data also stores the incoming acknowledgment messages (ACK), which an execution-thread transmits after processing a remote execution-queue, and the commit protocol uses them.

Commitment Protocol Q-Store's design allows us to support two light-weight commitment protocols. We can commit a transaction as soon as its last operation has been

processed when using conservative execution. Alternatively, we can defer the commitment of all the transactions to the end of the batch when using speculative execution.

Note that the former approach requires additional implementation complexity to ensure that committed transactions do not read uncommitted updated from aborted transactions. The latter approach could cause a non-trivial increase in the latency at the client because all transactions are committed at the end. However, the latter approach also helps the system to amortize the cost of the commit protocol over a batch of transactions [31].

One of the key advantages of employing deterministic transaction processing protocols is that non-deterministic aborts are no longer possible (e.g., aborts induced by concurrency control algorithms). Therefore, there no need to rely on *costly commit protocols*, such as 2PC.

For speculative execution in Q-Store, the commit protocol commits the whole batch after all the execution-queues are processed. On completing the execution of an execution-queue, the worker thread sends an ACK message notifying the planner's node about it. When the planner's node receives the ACK message, it updates the batch meta-data associated with the remote execution-queue. Further, Q-Store requires the local execution-threads to directly update the batch meta-data. When all the local execution-queues are executed and remote execution-queues are acknowledged, the planner node starts the commit stage for the planned transactions.

To commit a particular transaction, we check if all of its fragments' dependencies are resolved. If so, the transaction is committed. Otherwise, the transaction needs to be aborted, and the rollback process is started. During rollback, the speculative dependency path is walked, and dependent transactions are aborted. Note that, in the conservative execution, there are no speculative dependencies, and there are no cascading aborts.

3.6 Evaluation

In this section, we present an extensive evaluation of Q-Store. We implement our techniques in ExpoDB [2], [23]. We compare the performance of Q-Store's speculative execution

with the following concurrency control techniques. The conservative execution's performance evaluation and analysis remain future work.

- NO-WAIT: A representative of pessimistic protocols. A two-phase locking (2PL) variant that aborts a transaction if a lock cannot be acquired [34].
- TIMESTAMP: A basic time-ordering protocol [34] that is a representative of time-ordering concurrency control protocols.
- MVCC: An optimistic concurrency control protocol that relies on maintaining multiple versions of the accessed records. We select MVCC as representative of multi-version concurrency control protocols.
- MaaT: An optimistic concurrency control protocol [64] that is a representative of optimistic concurrency control protocols.
- Calvin: A deterministic transaction processing protocol [12].

We use a range-based partitioning instead of the original hash-based partitioning used by [13].

Cluster Setup We use a total of 32 Amazon EC2 instances for all experiments (16 server nodes and 16 client nodes). The instance type *c5.2xlarge*, which has 16GB of RAM and 8 vCPUs. We use Ubuntu 16.04 (xenial), GCC 5.4, Jemalloc 4.5.0 [65], [66] and compile our code with -O2 compiler optimization flag. We pin threads to cores to reduce the variance from the operating system scheduling and the effect of the caching system. Each dedicates 4 threads as worker threads, and 4 as communication threads. For Calvin, 2 out of the 4 worker threads are dedicated to sequencing and scheduling tasks. Each client node maintains a load of $10K$ active concurrent transactions.

Workloads We use two common macro-benchmarks for our evaluation. The first one is YCSB [38]. YCSB is representative of web applications used by YAHOO. The YCSB benchmark is modified to have transactional capabilities by including multiple operations per transaction. Each operation can be either a `READ` or a `READ-MODIFY-WRITE` operation. The benchmark consists of a single table that is partitioned across server nodes, and each

Table 3.1. Workload configurations parameters. Default values are in parenthesis.

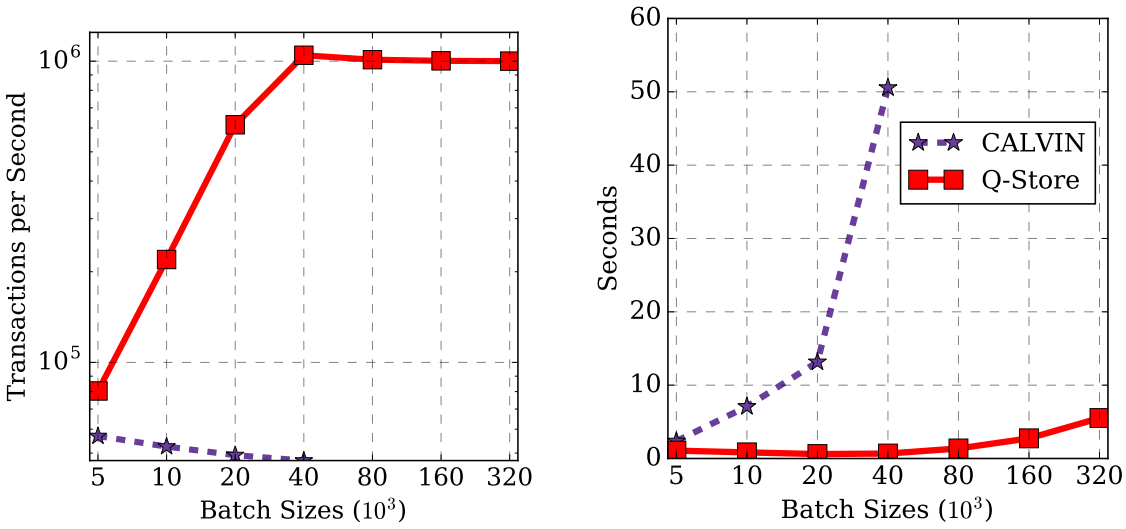
Parameter Name	Possible Parameter Values
Common parameters:	
% of multi-partition txns.	$1\%, 5\%, 10\%, 20\%, (50\%), 80\%, 100\%$
YCSB Workloads:	
Zipfian's theta	$(0.0), 0.3, 0.6, 0.8, 0.9, 0.99$
% of write operations	$0\%, 5\%, 20\%, (50\%), 80\%, 95\%$
Operations/txn.	$2, 4, 8, 12, (16)$
Partitions accessed/txn.	$2, 4, (8), 12, 16$
Server nodes counts	$2, 4, 8, (16)$
Batch sizes	$5K, 10K, 20K, 40K, (80K), 160K, 320K$
TPC-C Workloads:	
% of Payment txn.	$0\%, 50\%, 100\%$

node hosts 16 million records. The benchmark can be configured to capture various workload characteristics.

We also experiment with workloads based on the industry-standard TPC-C [39]. The TPC-C benchmark simulates a wholesale order processing system. There are 9 tables and 5 transaction types in this benchmark. All tables are partitioned across server nodes, where a partition can host one or more warehouses. Similar to previous studies in the literature[7], [13], we focus on the two main transaction profiles (`NewOrder` and `Payment`) out of the five transaction profiles, which correspond to 88% of the default TPC-C workload mix [39].

We report the average of 3 trials where each experiment trial runs for 120 seconds, and we ignore the measurements of the first 60 seconds, as it is used as a warm-up period. All reported measurements are observed by the client-side; thus, they are reflective of practical settings. Table 3.1, shows the various configuration parameters we used in our evaluation. Unless mentioned otherwise, we employ the default values.

Our experimental evaluation focuses on answering the following questions: (1) How does batch size affects the performance of batch-based distributed transaction processing systems (e.g., Calvin and Q-Store)? How do these systems handle high-volume workloads with large batches of concurrent multi-partition transactions? How do the following workload characteristics impact the performance of distributed transaction processing protocols: (a) the contention induced by data access skew; the percentage of multi-partition transactions in the



(a) Throughput

(b) 99^{th} Percentile Latency

Figure 3.5. Impact of varying batch sizes on the system throughput and 99^{th} percentile latency of deterministic systems.

workload; (b) the percentage of update operations in each transaction; (c) the transaction size (i.e., the number of operation per transaction); (d) the number of partitions accessed per transaction, and; (e) the transaction profiles? (3) How do these transaction protocols scale with respect to the number of nodes in the cluster?

3.6.1 YCSB Experiments

The YCSB benchmark is versatile, and we use it to answer many of the questions related to sensitivity factors. We start by studying the impact of batch sizes for protocols that rely on batching.

Impact of Batch Sizes Using the YCSB benchmark, we first study the impact of batch size on protocols that rely on batching, such as Calvin and Q-Store. The current implementation of Q-Store uses a count-based batch demarcation mechanism. On the other hand, the original Calvin implementation uses a time-based mechanism. For this set of experiments to be meaningful, we modified Calvin to use a count-based batch demarcation mechanism and make it stand on the same ground as Q-Store. We use the default parameters

and varying the batch size from 5K to 320K. The results are shown in Figure 3.5. Compared to Calvin, Q-Store scales very well as we increase the batch size up to $80K$.

Moreover, Calvin's throughput is very low because both the sequencing layer and the scheduling layer are single-threaded per node. With a large number of transactions per batch, those layers act as a bottleneck for the system. These results also show that Q-Store's architecture can utilize computing and network resources more efficiently. Beyond $80K$, the throughput of Q-Store plateaus as transaction processing becomes CPU-bound, and the latency starts to increase because worker threads take more time to process large batches. Calvin cannot handle large batches as transaction latency values exceed the experiment period. Remarkably, at $40K$ batches, Q-Store demonstrates an improvement of 22.1× the throughput of Calvin and an order of magnitude lower latency.

The most significant insight for Q-Store is that for large deployments (e.g., here, we have a total of 64 worker threads distributed over 16 server nodes), we need more work per thread to ensure efficient transaction processing and to hide the latency. Q-Store can handle large batches of concurrent transactions while keeping the latency low.

The presented results indicate that Q-Store is efficient in terms of performing useful work locally. The bottleneck is in the communication protocol, which is expected because the network is slower than local communication.

In the remaining experiments, we use the original time-based batch demarcation mechanism for Calvin and use their reported parameter of $5ms$ [13]. We observe that with $5ms$ time-based batch demarcation, Calvin produces batches of size 160 per node approximately.

Variable Contention In this set of experiments (Figure 3.6), we vary the Zipfian skew factor θ from 0.0 (uniform) to 0.99 (extremely skewed). As θ approaches 1.0, the data access becomes more skewed within a partition, but the partitions are chosen uniformly per transaction. In other words, each partition receives uniform access, but the record access within the partition is skewed. It is possible to use a Zipfian distribution for partitions as well, but that would not measure the performance of how each node is dealing with skewness. Further, in such a case, mostly one node is active while the remaining nodes are idle most of the time. We use a 50% multi-partition workload such that the 16 operations in a transaction randomly access exactly 8 partitions.

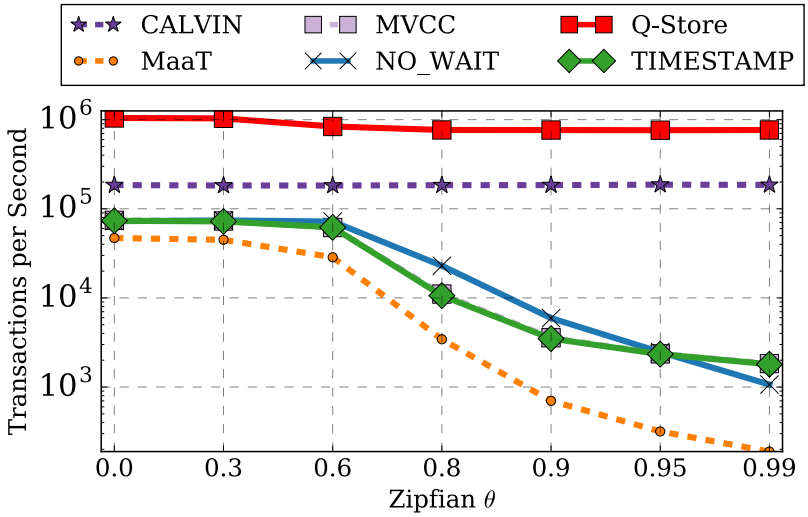


Figure 3.6. Impact of varying the data access skewness parameter θ of the Zipfian distribution on systems throughput (log scale).

Both Calvin and Q-Store perform better because they both avoid the cost of the two-phase commit protocol (2PC). However, Q-Store achieves up to 6× better throughput. The first reason for that is queue-oriented execution and communication. Q-Store sends a queue of ordered operations that belong to several concurrent transactions to remote nodes. Thus, Q-Store ensures a more efficient communication.

Since different threads execute queues in parallel, Q-Store exploits intra-transaction parallelism (both within a node and across nodes) better than Calvin. For Calvin, the level of contention does not affect its performance because the bottleneck is in the sequencing and scheduling layer. Note that Q-Store's throughput degrades slightly under high-contention (i.e., beyond $\theta = 0.6$) due to the imbalance in the size of execution queues.

The throughputs for non-deterministic protocols are low because they require a costly 2PC protocol for committing each transaction. As the contention increases, the abort rates also increases, which lowers their performance even more. When transactions abort, they are retried using a random back-off period. Under high-contention, transactions may abort multiple times, which effectively increases the latency per transaction, which lowers the throughput. Remarkably, Q-Store achieves nearly two orders of magnitude better system throughput under high-contention in comparison to non-deterministic protocols.

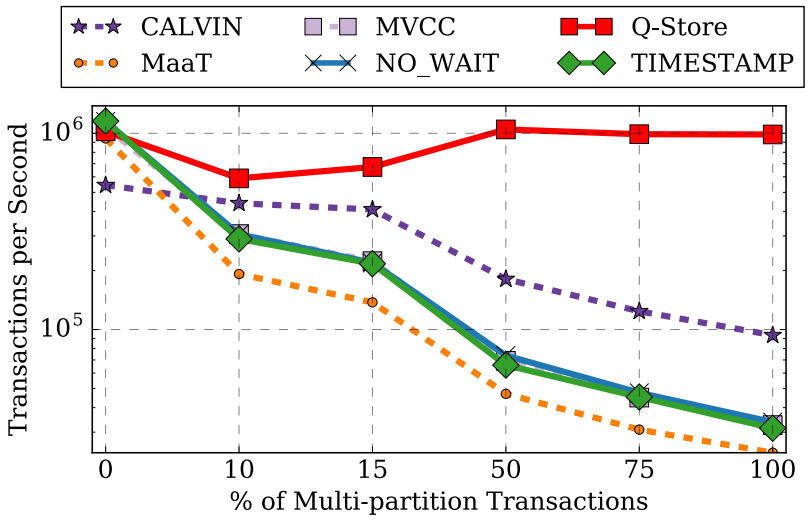


Figure 3.7. Impact of varying the percentage of multi-partitions transactions in the workload on the system's throughput.

Varying multi-partition transactions rate Now, we focus on the impact of multi-partition transactions in the workload. We vary the percentage of multi-partition transactions in the workload from 0% (single-partition transactions only) to 100% (multi-partition transactions). We fix the values of other parameters to the default values. The results shown in Figure 3.7 are for low contention (i.e., $\theta = 0.0$). Note that in comparison to Figure 3.6, there is no noticeable difference in the throughputs of the protocols with single-partition transaction workloads, except for Calvin.

Non-deterministic protocols do not need to perform 2PC, which allows them to avoid 2PC's cost. When the rate of multi-partition transactions increases, non-deterministic protocols incur the overhead of 2PC to ensure serializable execution, and thus, their throughput decreases. Thus, our results validate previously published results (e.g., [13]), which illustrate the poor performance of non-deterministic protocols.

Despite the deterministic nature of Calvin, its throughput also decreases as the rate of multi-partition transaction increases. Calvin needs to send a given transaction to all participants and waits for their responses before scheduling the next conflicting transaction. This approach increases the communication overhead per transaction and negatively affects the performance of Calvin. Unlike Calvin, Q-Store is not sensitive to multi-partition transactions.

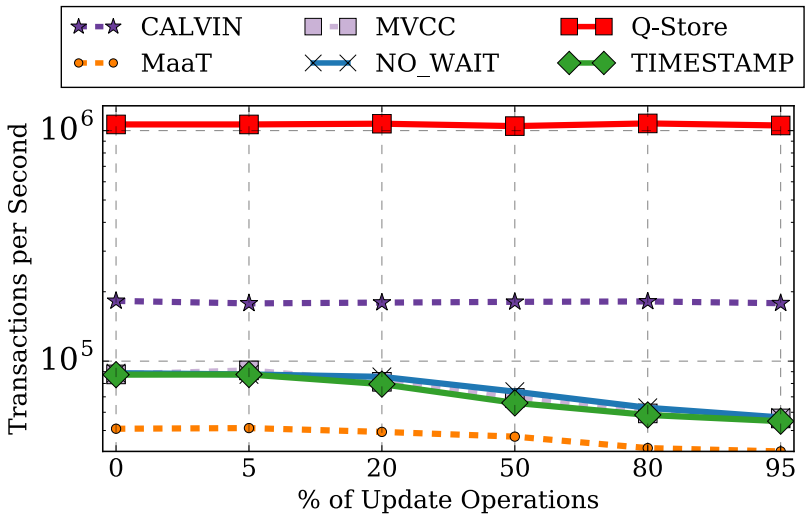


Figure 3.8. Impact of varying the percentage of update operations in the workload on the system's throughput.

In addition to avoiding 2PC overhead, it has minimal communication overhead. Q-Store communicates only a minimal number of execution queues between partitions, which contain scheduled operations of several transactions. Thus, it effectively reduces the communication overhead per transaction. Q-Store outperforms Calvin's throughput by up to 10.6×.

Vary the percentage of update operations In the following experiments, we study the impact of the percentage of the update operations on the transaction processing performance. In previous experiments, we used a value of 50%, which means that 8 out of 16 operations are updating the database in each transaction. To study this factor, we vary the percentage of update operations from 0% (read-only operations) to 95%. We fix the remaining parameters to their default values. Note that increasing the rate of update operations increases the contention on records (e.g., exclusive locks induce record contention).

Figure 3.8 shows the result of varying the percentage of update operations. The results show that neither Q-Store nor Calvin are sensitive to this factor. Calvin employs deterministic locking to avoid aborting transactions unnecessarily while Q-Store executes operations according to their order in a given queue. In other words, for Q-Store there is no difference between the *read* or *update* operations as Q-Store executes each operation in-order, which eliminates any sensitivity to this factor. With non-deterministic protocols, we observe that

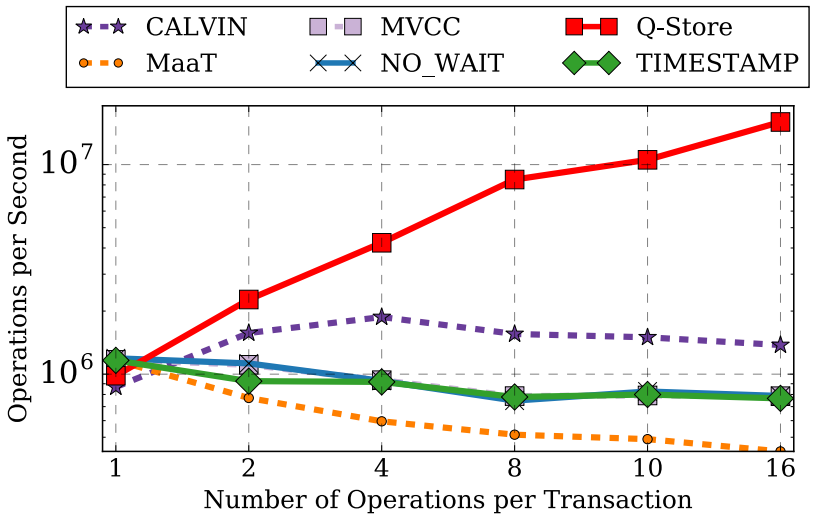


Figure 3.9. The impact of varying the number of operation per transaction on system's throughput. We force each operation access a different partition. This results is for low contention $\theta = 0.0$.

the abort-rate increases as the contention increases due to more update operations in the workload. For NO-WAIT, MaaT, TIMESTAMP, and MVCC, the abort-rates are up to 41%, 19%, 7%, and 6%, respectively, at 95% update rate.

When a transaction is read-only, there is no need to perform 2PC, but participants still need to communicate messages to finalize the running transaction. As the transaction involves more update operations, the overhead of 2PC protocol becomes more substantial, which negatively affects the performance of non-deterministic protocols that rely on 2PC as their atomic commitment protocol. Notably, Q-Store shows an improvement in its system's throughput by up to 5.9× and 17.1× over Calvin and MVCC (the next best non-deterministic protocol), respectively.

Vary the number of operations per transaction Now, we experiment with varying the number of operations per transaction. We set the percentage of multi-partition transactions to 50%, and force each transaction to access the same number of partitions as its number of operations. For example, if a transaction has 4 operations, the number of partitioned accessed by that transaction is also 4. However, each partition has the same probability of access by any operation, and we do not force operations to be remote.

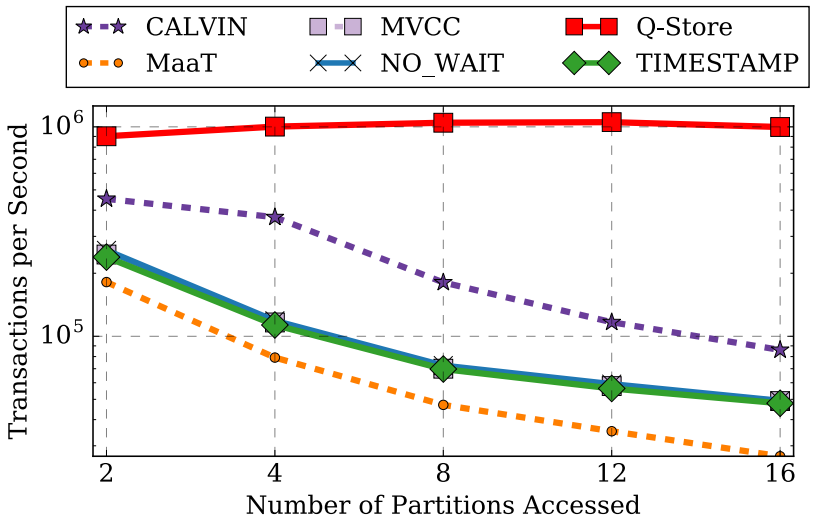


Figure 3.10. The impact of varying the number of partitions accessed by each transaction on the system's throughput.

This experiment aims to capture execution and communication overheads as transactions become larger. For non-deterministic protocols, as the number of operations increases, the cost of 2PC increases because it is more likely that more nodes need to participate in the commitment protocol. Calvin performs better than other non-deterministic protocols, but its performance does not scale with larger transactions. Q-Store, on the other hand, scales well as the number of operations per transaction increases. With 16 operations per transaction, Q-Store's performance reaches a remarkable throughout of nearly 16 million operations per second. These numbers are 12× and 20× better than those for Calvin and NO-WAIT, respectively, as shown in Figure 3.9. These gains are due to the proposed efficient queue-oriented execution and communication. For Q-Store, the number of queues communicated is constant (but their sizes may vary) while the other protocols exchange messages for remote operations, which increases the overall communication overhead.

Vary partitions per transaction In Figure 3.10, we show the results for varying the number of partitions accessed by transactions having 16 operations. We use uniform data access, which leads to a low contention workload. By having uniform data access, the effect of contention is negligible, which can help us to examine the communication costs. As we increase the number of partitions accessed by a transaction, the overhead of committing this

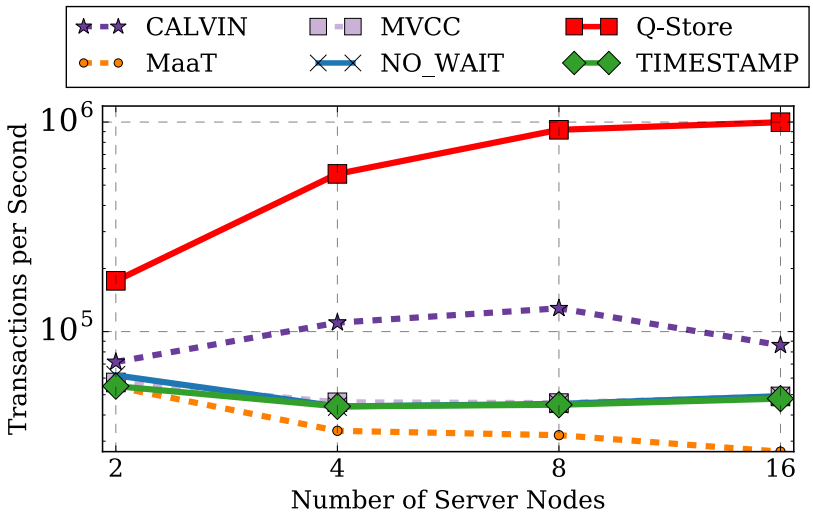


Figure 3.11. Throughput scalability results while varying the number of server nodes.

transaction increases because the commitment involves agreement of more participants per transaction. This issue is mainly a problem for non-deterministic protocols as the participants need to agree on the order for operations. As the number of participants increases, more coordination is required to commit each transaction.

While Calvin eliminates the overhead of 2PC, it still suffers from increasing the number of partitions accessed per transaction. The reasons for that are: (i) it needs to send the transactions to more participants, and (ii) it needs to wait for acknowledgments from more participants before declaring a transaction as committed. This communication overhead increases as the number of partitions accessed increases. In contrast, Q-Store demonstrates its insensitivity to this factor and achieves a throughput of around a million transactions per second despite the increase in the number of partitions accessed per transaction. Since the workload is uniform, the number of partitions accessed affects only the sizes of remote execution queues, and there is no increase in the number of communicated execution queues.

Scalability For all previous experiments, we have used 16 servers. In this set of experiments, we vary the number of nodes to evaluate the scalability. We set the percentage of multi-partition transactions to 50%, and force each transaction to access all available partitions. Figure 3.11, shows that Q-Store scales well as the number of server nodes increases in the cluster, achieving over 1 million transactions per second at 16 server nodes.

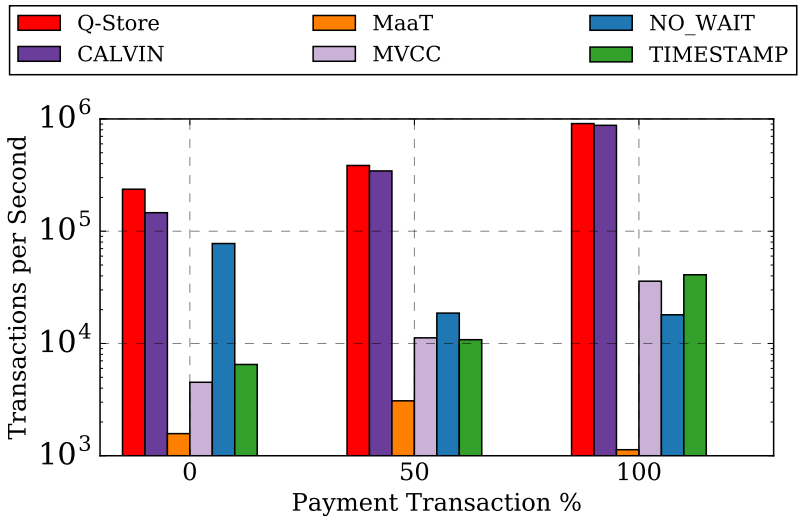


Figure 3.12. The impact of different TPC-C transaction mixes on the system's throughput. 15% multi-partition transactions is used.

Other approaches do not scale due to the overhead of multi-partition transactions. Calvin's performance cannot scale because of the single-threaded pre-execution phases, while non-deterministic protocols do not scale due to the increased overhead of 2PC.

3.6.2 TPC-C Experiments

We also evaluate Q-Store with workloads based on the industry-standard TPC-C benchmark. For this set of experiments, we use a total of 16 server nodes, with 4 warehouses per server. Hence, the total number of warehouses is 64. We use three workloads: 100% NewOrder-transaction workload, 50% Payment and 50% NewOrder transactions workload mix, and finally 100% Payment-transaction workload. We use the standard rate of 15% of the payment transactions coming from remote customers as the multi-partition transaction rate, for all the transactions in the workloads. We also restrict the number of partitions accessed to *two* even for NewOrder transactions.

The results are shown in Figure 3.12. Both deterministic systems Calvin and Q-Store significantly outperform other algorithms by a significant margin due to their use of 2PC. Q-Store outperforms Calvin by up to 1.8×. Remarkably, Q-Store outperforms NO-WAIT, which is the best performing non-deterministic protocol, by up to 55.2×. NO-WAIT suffers from

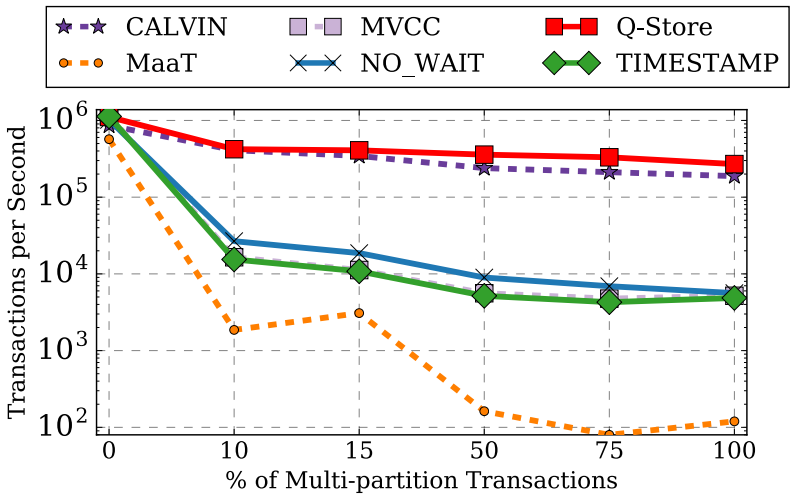


Figure 3.13. Varying the percentage of multi-partition transaction with equal ratios of Payment and NewOrder transactions.

high abort rates due to contended warehouse records (to avoid deadlocks) and the overhead of 2PC for multi-partition transactions. On the other hand, Q-Store eliminates the overhead of 2PC and execution-induced aborts.

The second set of experiments that use TPC-C workloads study the effects of multi-partition transaction rates (Figure 3.13). The transaction profiles in TPC-C are more complicated than their YCSB counterparts. It involves data dependencies among operations, which can reduce the performance of Q-Store. For example, in the NewOrder transaction, many operations require the new value of the OrderId, which is updated by the same transaction. Our current implementation creates an execution-queue per warehouse, which serializes all operations accessing records belonging to a given warehouse. Despite this unfavorable data partitioning scheme, Q-Store's throughput still outperforms Calvin's throughput. [1]

3.7 Related Work

Research on distributed transaction processing systems started several decades ago. One of the key challenges in distributed transaction processing is managing the execution of concurrent transactions such that they produce serializable execution histories. Bernstein and

[1]↑ For TPC-C like workloads, unlike Calvin, Q-Store's performance can be further optimized by further splitting execution-queues and exploit parallelism instead of serializing operations per warehouse. However, such optimization is beyond the scope of this chapter, and we leave it to future work.

Goodman [34] give a comprehensive overview of distributed concurrency control techniques. In this section, we cover some of the recently proposed distributed transaction processing systems and transaction processing techniques that are mostly related to Q-Store. We categorize them as follows.

Non-deterministic Transaction Processing When a transaction updates multiple partitions of the distributed database, there is a need for a commit protocol to ensure that the updates are consistent across all the partitions because nodes may arrive at distinct order of execution for the transaction operations. As a result, aborts may occur non-deterministically. The two-phase commit protocol is typically used to resolve this problem, but naive implementations of 2PC suffer from costly overheads, which negatively impact the system performance. Therefore, many optimizations for 2PC have been proposed (e.g., [64], [67]–[69]) while preserving the non-deterministic nature of execution. However, due to this non-determinism, these systems suffer from execution-induced aborts and cannot eliminate the overhead of 2PC [63]. In contrast to these approaches, Q-Store processes transactions deterministically and eliminates the overhead of 2PC and non-deterministic aborts during execution.

Eliminating Multi-partition Transactions Some proposed approaches avoid the cost of 2PC by avoiding the need to process multi-partition transactions. For example, G-store [70] allows applications to declare arbitrary groups of records and moves these groups to a single node to avoid the overhead of processing multi-partition transactions. In a similar spirit, LEAP [54] avoids the cost of 2PC by moving records accessed by a given transaction to a single node at run-time implicitly. Q-Store, on the other hand, embraces multi-partition transactions, and deterministically orders operations into execution-queues; thus avoiding the need for a 2PC protocol.

Deterministic Transaction Processing Deterministic approaches to transaction processing showed great potential in the academic research literature and even had commercial offerings, e.g., [56], [71]. For single-partitioned workloads, H-Store[14] uses single-threaded serial execution per partition. For workloads having multi-partition transactions, H-Store provides limited concurrency by employing a coarse-grained locking mechanism that locks all the partitions prior to the start of a transaction. Jones et al. [52] studies the application

of speculative concurrency control to multi-partition transactions in H-Store, which allows transactions to read uncommitted updates of transactions that are performing distributed commitment protocol. Unlike H-Store, Q-Store does not lock partitions to produce a serializable execution for operations of multi-partition transactions. Instead, Q-Store creates execution-queues that capture the serializable order of conflicting operations, and it assigns these execution-queues to worker threads. After that, each worker thread executes its assigned execution-queues according to the pre-determined priority of execution-queues, which allows Q-Store to maintain its high performance despite the multi-partition workloads.

In Gargamel [49], a single dedicated load-balancing node pre-serializes (using static analysis) possibly conflicting transactions before their execution. The load-balancing node can easily become the bottleneck for the system. Unlike Gargamel, Q-Store is centered around the notion of priority and exploits multiple nodes for planning.

Calvin [12], [61] uses determinism to eliminate the cost of two-phase-commit protocol when processing distributed transactions. T-Part [61] relies on the same system architecture of Calvin, but its scheduling layer constructs transaction dependency graphs to reduce the stalling of worker threads. There are fundamental architectural differences between Calvin and Q-Store. The planning phase performs the same functionality as the two-step (sequencing and scheduling) pre-processing phases, but *in parallel*, and the execution phase of Q-Store does not rely on any locking mechanism and employs a queue-oriented (speculative and conservative) processing design. Additionally, in contrast to Calvin, which assigns a transaction to a worker thread for processing, Q-Store assigns an execution-queue to a worker. Because of this thread-to-transaction mapping, Calvin cannot exploit intra-transaction parallelism opportunities within a single node.

Intra-transaction Parallelism Most transaction processing systems perform a thread-to-transaction assignment, which makes these systems unable to exploit intra-transaction parallelism efficiently. Several research studies proposed techniques for exploiting this kind of parallelism in centralized environments (e.g., [2], [11], [26], [28]). Q-Store goes beyond these proposals and exploits intra-transaction parallelism within and across nodes in the context of distributed transaction processing.

3.8 Conclusions

We presented Q-Store, which efficiently processes distributed multi-partition transactions via queue-oriented priority-based execution model. We present a formalization of our system and describe its design and implementation. We perform an extensive evaluation of Q-Store using different workloads from standard benchmarks (that is, YCSB and TPC-C). We demonstrate that Q-Store, consistently and significantly achieves higher performance than existing non-deterministic and deterministic distributed transaction processing systems. We experimentally demonstrate that Q-Store out-performs the state-of-the-art deterministic distributed transaction processing protocol by up to 22.1× with YCSB workloads. Against non-deterministic distributed transaction processing protocols, Q-Store achieves up to two orders of magnitude better throughput with YCSB workloads, and up to 55× with TPC-C workloads.

4. HIGHLY AVAILABLE QUEUE-ORIENTED SPECULATIVE TRANSACTION PROCESSING

4.1 Introduction

Cloud providers continue to provide a virtual computing infrastructure that provides a higher amount of main memory and virtual CPU cores. Currently, for instance, Amazon Web Services provides virtual instances configurations that are equipped with up to 448 vCPUs, $24TB$ of memory, and $100Gbps$ network connectivity.[1] Therefore, there is a growing demand for utilizing this modern computing infrastructure efficiently.

Many deterministic database systems are proposed in the research literature to utilize modern computing infrastructures more efficiently (e.g., [12], [14], [63]). Recent proposals of distributed deterministic transaction processing (DTP) systems demonstrated significant improvements over systems using traditional transaction processing techniques (e.g., 2PL/OCC+2PC). While distributed DTP systems have shown significant improvements in transaction processing performance, many database applications require high availability. For example, users of online banking applications desire that it is available 24×7. Furthermore, cloud providers' service level agreements promise at least four nines (i.e., 99.99% availability).[2] Database replication for traditional transaction processing protocols is well-studied (e.g., [72]–[74]). In contrast, the problem of ensuring high availability via replication in distributed DTP systems has received less attention from the research community, and this aspect of distributed DTP systems has not been analyzed and evaluated well.

We consider database systems where the database state can be partitioned and distributed across multiple nodes (e.g., [12], [14]–[16], [22], [55], [75]). Furthermore, the partitioned database state is replicated for high availability. With deterministic transaction processing, the replication is simplified because transaction histories are deterministic and strictly serializable. Strict serializability implies that transaction execution of conflicting transactions follows a *single* order across all partitions and replicas. By requiring that the predetermined order is followed during execution and in the replicated state, the replication

[1]↑https://aws.amazon.com/ec2/instance-types/high-memory/
[2]↑https://azure.microsoft.com/en-us/support/legal/sla/mysql

process is simplified because the new database state is guaranteed to be equivalent due to deterministic execution. Thus, the key challenge with replication in distributed DTP systems is the negative impact of performing replication on the transaction processing performance.

To address this challenge, we build on our highly efficient queue-oriented transaction processing paradigm [2], [50], [76]. In our earlier work QueCC[2], we addressed the issue of the overhead of exiting concurrency control techniques under high-contention workloads and demonstrate that speculative and queue-oriented transaction processing can improve the system's throughput by up to two orders of magnitudes over the state-of-the-art centralized (non-partitioned) transaction processing systems. In Q-Store[50], we improve the efficiency of distributed and partitioned DTP systems by employing queue-oriented transaction processing techniques and demonstrate up to 22× better performance.

In this chapter, we propose a generalized framework to analyze the design space of distributed and replicated deterministic transaction processing systems and extend QueCC and Q-Store with replication support for high availability. Based on the proposed framework, we propose a primary-copy approach and perform eager, speculative, queue-oriented replication to mitigate the overhead of replication in distributed DTP systems. Our approach amortizes the cost of replication and transaction processing over batches of transactions and processes these batches in parallel on a replicated clusters of server nodes. Furthermore, we exploit the fact that deterministic transaction execution and replication in DTP systems are independent, which allows us to either fully or partially hide the cost of replication while ensuring safe and strictly serializable transaction execution.

Our contributions in this chapter can be summarized as follows:

- we propose a generalized framework for DTP systems, a unified replication API for DTP systems, and apply the framework on three systems from the literature (Section 4.2);
- we design QR-Store, a highly available queue-oriented and replicated transaction processing system version of Q-Store (Section 4.3);
- we prototype QR-Store and propose optimization techniques to improve the performance of state-of-the-art in queue-oriented deterministic transaction processing (Section 4.4);

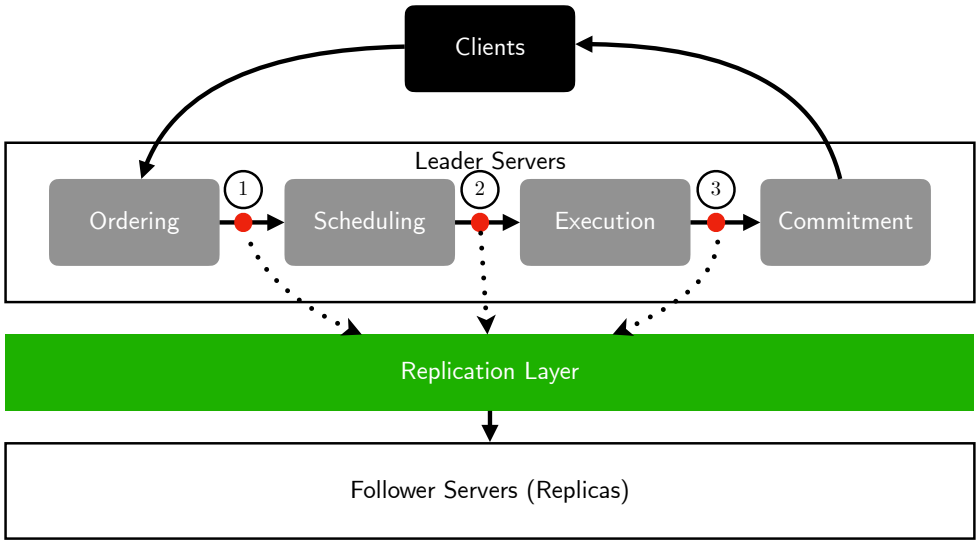


Figure 4.1. Generalized Deterministic Transaction Processing Framework

- we extensively evaluate QR-Store using standard benchmarks such as YCSB (Section 4.5).

4.2 A Generalized Replication Framework

We propose a generalized DTP framework. In Figure 4.1, we illustrate a simple framework that adopts the client-server architecture. The system is composed of a set of clients that sends transactions for processing to a set of servers. Clients receive commitment responses from servers when their submitted transactions are committed to the database. The transaction processing workflow by a leader set of servers is composed of four generic stages for processing transactions deterministically with strict serializability. These steps are *ordering*, *scheduling*, *execution*, and *commitment*. The processing work in each stage can be done in a parallel and distributed fashion to improve the system's performance (e.g., by multiple worker threads deployed on a set of machines). It is important to note that DTP systems use batching to improve the throughput performance of the system.

A DTP system ensures strict serializability of transaction histories by predetermining the *order* of transactions execution/commitment before scheduling them for execution. In the *scheduling* stage, the scheduling algorithm needs to guarantee that the execution and the

commitment of the transactions do not violate the predetermined order. In between stages, we define replication points (red circles in Figure 4.1). These are points in the transaction processing workflow where replication can be done. The output of a stage can be replicated using the replication layer to achieve system high availability. A set of follower servers (replicas) get the replicated output from a stage and proceed to use it as an input to the next stage.

In our framework, the replication layer is a logical layer. One implementation approach is by using a shim that interacts with a replicated coordination service such as Zookeeper [77] and etcd [78] or publish-subscribe systems like Kafka[79]. In this approach, the service serves as a physical middleware between the leader set of servers and the follower set of servers. Existing work uses Zookeeper in their prototype implementations (e.g., [12]), but Zookeeper is not designed for this purpose. In our experiments, we observed that Zookeeper could not handle the replication load when the replication request rate is high. Therefore, using a service like Kafka appears to be a better option, and we plan to study that in future work.

Another implementation approach for the the replication layer is having the shim implements a protocol such as Paxos [80], Viewstamped replication [81], or Raft[82] directly. This integrated approach has a lower overhead (no need for additional dedicated nodes for the replication layer). However, it involves a more tight integration with the DTP system and is more complex to realize.

To realize both approaches in a generalized way, we introduce a simple API that abstracts away the complexity and hides the details behind the underlying implementations. The API is compromised of two simple functions replicateData and receiveData. More details about this API are presented in Section 4.4.

Our proposed framework is general enough to admit existing work on deterministic transaction processing systems as specialized implementations. We discuss three case studies to illustrate the applicability of the proposed generalized framework to provide a unified framework to understand DTP systems.

4.2.1 Case Study: Calvin

Calvin [12] is one of the first DTP systems that supports replication. In Calvin, the *ordering* stage performs epoch-based batching of transactions and it is called *sequencing*. Calvin uses the replication point that follows the *ordering* stage and replicates batches of sequenced transactions. In the *scheduling* stage, Calvin uses a deterministic locking algorithm to schedule transactions for execution. In deterministic locking, the order of lock acquisition follows the predetermined transaction order by the sequencing layer. Calvin is a distributed DTP system and requires the use of a distributed commit protocol (CP) in the *commitment* stage. However, it avoids using the traditional heavyweight two-phase commit protocol and uses a lightweight CP that exploits deterministic execution. Transactions in Calvin commit when all the operations of the distributed transactions complete. The CP aborts transactions when the transaction has a logic-induced abort, and it is aborted deterministically across all partitions and replicas. In the absence of a logic-induced abort, transactions are committed, and the commit response is sent to the clients by the sequencing node that originally received the transaction and sequenced it.

Calvin's original proposal [12] proposed replicating the output of the *ordering* stage. However, based on our framework, it is possible to use other replication points. For example, the updated records in the execution stage can be logged and replicated before commitment.

4.2.2 Case Study: Q-Store

Q-Store [50] is also another distributed DTP system, but unlike Calvin, it combines the *ordering* stage and the *scheduling* stage into a single parallel stage called *planning*. The *execution* stage uses the concept of execution queues (EQs) of operations as an execution primitive while Calvin uses the concept of a transaction as an execution primitive. Q-Store focuses on distributed transaction processing without replication. The *planning* stage maps batches of transactions to execution queues tagged with execution priorities. The execution stage executes them based on their priorities, and the commitment stage maps them back to transactions and sends responses to clients.

Replication in Q-Store can use any one of the replication points. The first replication point occurs in the middle of planning and is similar to the replication in Calvin (i.e., replicating sequence batches of transactions). At follower nodes, the replicated sequence is planned into execution queues. Interestingly, if the first replication point is used, it is possible to have heterogeneous configurations of servers. For example, a group of servers can follow Calvin's DTP approach while the others can follow Q-Store's.

When using the second replication point, which is after the *planning* stage, a novel replication scheme emerges. Because the execution primitive of Q-Store is a set of execution-queues (a.k.a. plans), the set is replicated to follower servers, and the follower servers can take the replicated plans and use them in the next stages. With this approach, Q-Store is also required to replicate transaction contexts so that in the commitment stage, the replicas can map execution queues back to transactions for commitment.

Q-Store can also use the third replication point, which also introduces yet another novel replication scheme. In this case, instead of creating traditional logs, Q-Store creates plans of execution queues containing write-only operations of updated records. When replicated successfully, it is fed to the execution stage at the replicas, and no specialized stage is needed to process the replicated plans. Furthermore, only the last write operation on the record needs to be inserted in the write-only execution queues.

4.2.3 Case Study: QueCC

QueCC [2] is a single node DTP system that is designed and optimized for multi-socket, many-core machines. QueCC uses the same concepts as Q-Store in terms of having *planning* and *execution* stages, but all stages are parallel but not distributed by design. QueCC can be extended to become a replicated DTPS. In this case, the leader server set contains only a single node that contains the entire database state, and its state is replicated using the replication layer. Compared to Q-Store, QueCC does not exploit partitioning and horizontal scalability; however, it can scale vertically by using more cores. Furthermore, it is possible to have heterogeneous hardware configurations for replicas where replicas don't have the same hardware specifications as the leader node.

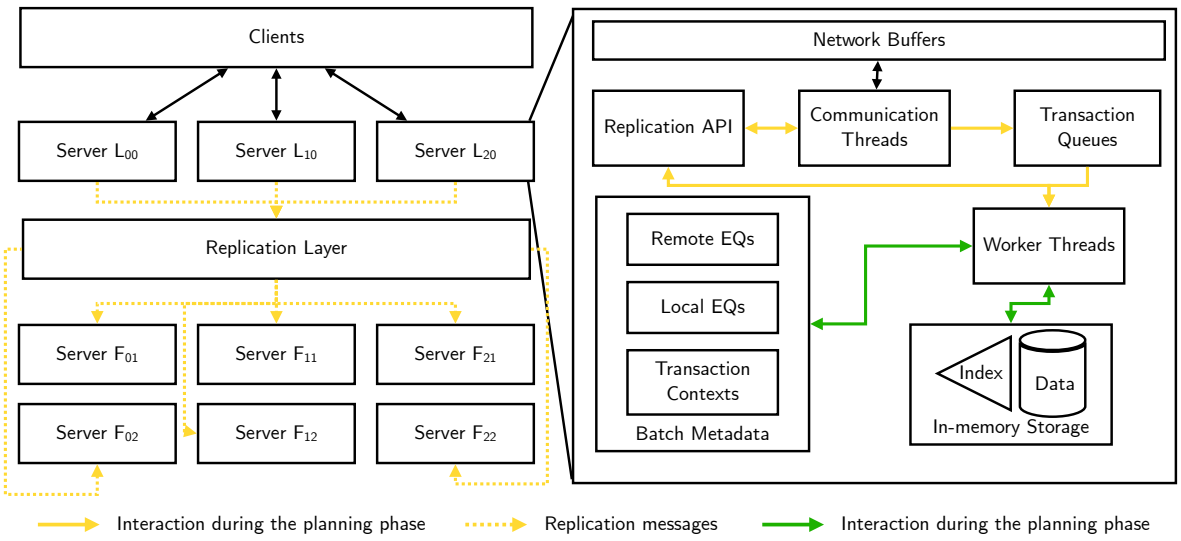


Figure 4.2. QR-Store System Architecture

4.3 Queue-oriented Transaction Processing

Based on the generalized framework described in Section 4.2, we focus on designing replication schemes for Q-Store and study their impact on the system's performance. We build QR-Store which is a replicated version of Q-Store and give some overview of QR-Store.

4.3.1 QR-Store Architecture

As a distributed DTP system, QR-Store runs on a cluster of nodes. Each node holds a partition of the database. It supports processing multi-partition transactions where a transaction may access records from different partitions. Each partition is replicated independently with a replication factor rf. For example, if $rf = 2$ for partition p_0, then the system has three nodes hosting p_0. One of them is the leader node, while the others are followers. On the left side of Figure 4.2, we show an example system architecture with three partitions and a replication factor of three (i.e., $rf = 2$). Visually, horizontal grouping of nodes implies a cluster of QR-Store nodes comprising a full replica of the distributed database instance, while vertical grouping implies replication groups. For example, nodes L_{00}, L_{10}, and L_{20} form a cluster instance of QR-Store, while nodes L_{00}, F_{01}, and F_{02} form a replication

group. Note that replication messages are communicated within a replication group only, while all other messages related to processing transactions are communicated within a cluster instance. This communication scheme ensured minimal communication among nodes in QR-Store.

On the right side of Figure 4.2, we show the key components internal to a server node. Each node receives client transactions that are processed by communication threads into a set of client *Transaction Queues*. *Worker threads* on each node process client transactions in two phases: *planning* and *execution*. Note that we consider the commitment stage as part of the execution phase. In the planning phase, worker threads create execution queues (EQs) that access a sub-partition of the node's partition. To facilitate scheduling of EQs during the execution phase in QR-Store, each worker thread in QR-Store tags its EQs with a priority value. This value can be static or dynamic, but we assume statically predetermined priorities. There are *remote* EQs and *local* EQs. Remote EQs are executed at remote nodes as transaction fragments in them access other remote partitions. In addition to EQs, transaction contexts are maintained, which captures transaction dependencies and other transactions metadata. EQs and transaction contexts are stored in the *Batch metadata*, which distributed shared data structure. Furthermore, worker threads in the leader set of nodes use the *Replication API* to facilitate replication of the Batch metadata to the replicas.

During the execution phase, worker threads execute and commit EQs based on their priorities. For example, say we have two EQs q_{i} and q_{j}. The fact that q_{i} can be either remote or local is orthogonal, and the same applies to q_{j}. Let $pr(q)$ denote the priority of an EQ q. QR-Store maintains a global execution invariant such that q_{i} is executed before q_{j} *if and only if* $pr(q_{\text{i}}) > pr(q_{\text{j}})$. Maintaining this global execution invariant with a cluster ensures a single global order of conflicting operations, which produce strict serializable histories. In Figure 4.2, yellow arrows depict interactions during the planning phase while green arrows depict interactions during the execution phase.

Algorithm 1 Planning phase

VARs: CTQ client transaction queue, C : nodes in the cluster instance, TC : transaction contexts data structures, s : status of the current node

```
1: function PLANBATCH(s, bid, p)
2:     B ← {}
3:     if isLeader(s) then
4:         while not B.ready() do
5:             m ← CTQ.pop()
6:             EQ ← planMessge(m, TC)
7:             B ← B ⊕ EQ
8:         end while
9:         return DELIVERBATCH(B)
10:    else
11:        return RECEIVEDATA((bid, p), (B, TC), DELIVERBATCH(B))
12:    end if
13: end function
14: function DELIVERBATCH(B)
15:    LEQ ← {q ∈ B| isLocal(q)}
16:    REQ ← {q ∈ B| isRemote(q)}
17:    setLocalEQs(LEQ)
18:    sendRemoteEQs(REQ, C)
19:    REPLICATEDATA((bid, p),(B, TC))
20: end function
```

4.3.2 Replicated Planning Algorithm

We start by describing the planning algorithm. Algorithm 1 presents pseudocode for the planning phase. To simplify our presentation, we assume the availability of some global variables.

CTQ is a variable for the queue holding client transaction. Communication threads push into this queue as they receive transaction messages from clients.

C is the set of nodes composing the cluster instance. For example, suppose a worker thread running in the planning phase calls PLANBATCH on server L_{00}. In this case, $C = \{L_{00}, L_{10}, L_{20}\}$.

TC is the data structure that holds the transaction contexts for each planned transaction and holds necessary transaction metadata (e.g., the number of operations and their dependencies).

The variable s is the status of the node running the worker thread. For instance, for server $\{L_{00}\}, s = L$, and for $\{F_{01}\}, s = F$. When $s = L$, isLeader(s) = true.

The PLANBATCH function is called by each worker thread in each server. Each thread starts by computing the batch identifier *bid*, which determines the order between batches created at different epochs. In our prototype implementation, we use monotonically increasing numbers for batch identifiers. Therefore, for a batch created at epoch 0, its batch identifier is also 0. p is the global priority value of the worker thread. As mentioned previously, these values can be either static or dynamic. The only requirement is that no two threads have the same priority. In our implementation, we use static priority values for nodes and worker threads.

Depending on the status of the server node, the planning phase takes two different routes. In case of a node being in the leader set, the worker thread follows Lines $4-9$, reads a message from CTQ, and *plans* the message (Line 6).

When planning a message, the read/write set of the transaction is analyzed. When the read/write set of a transaction includes access to a remote partition, a transaction fragment is created and is placed into a remote EQ destined to the node hosting the target partition. Thus, knowing the full read/write set and the record-to-partition mapping is necessary for planning. The planMessage function returns a set of EQs, and they are *merged* with previously planned EQs. Merging EQs means that transaction fragments accessing the same partition are inserted into the same EQ.

The call of B.ready() at Line 4 determines when the batch is *ready* for delivery. Once the batch is ready, it is delivered to the respective nodes (Lines $14 - 20$). Local EQs are set in the local partition of the Batch metadata distributed data structure. Remote EQs are sent to their respective nodes to be installed into the remote partitions of the batch metadata. Finally, (in Line 19) the planned EQs and the transaction contexts are replicated using the replication API (i.e., calls REPLICATEDATA) to the replica groups (e.g., for L_{00} they are replicated to nodes F_{01} and F_{02}).

When the PLANBATCH function call is made by a follower node, it calls the RECEIVEDATA (Line 11) and provides the DELIVERBATCH function as the `callback` function. This way when the replicated plans are received, the DELIVERBATCH is called to deliver the planned

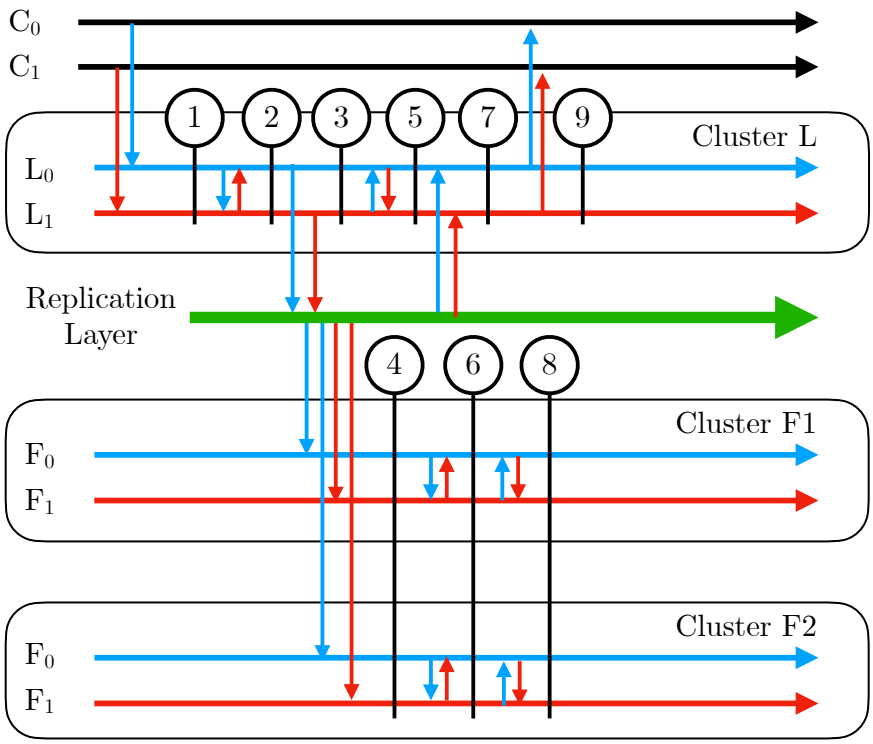


Figure 4.3. Speculative Execution and Replication Timeline Example

EQs to node in the replica cluster instance. Using Figure 4.2 as our example, node F_{01} delivers the replicated batch to nodes $P_1(F1)$ and $P_2(F1)$.

4.3.3 Speculative Queue-oriented Replication Protocol

As we described in Section 4.2.2, there are many possible replication schemes that can be used with QR-Store. We propose using the second replication point (from Figure 4.1), which is before the execution stage, to perform the replication of EQs and TC using the replication API. The EQs and TC are serialized into a byte string payload and replicated via the replication layer.

Speculative EQ Replication. We propose a queue-oriented speculative approach to replication. The replication is speculative because QR-Store speculates that the replication would be successful and proceed with the execution phase. The speculation is verified before the commitment stage. Thus, we effectively hide the EQs' replication latency by performing it concurrently with EQs' execution.

Figure 4.3 illustrates an example timeline (time goes from left to right) of the replicated transaction processing workflow. A client is associated with a server in the leader set that is considered the *home server* for that client. At the start (before ①), clients send transactions to home servers. In this example, C_0 sends transactions to server L_{00} (i.e., the blue partition) while C_1 sends transactions to L_{10} (i.e., the red partition). At ①, the batch is ready for delivery, and both leaders send their planned remote EQs to the other node. At ②, the leaders submit their replication requests to the replication layer (i.e., using REPLICATEDATA API call). Leader nodes start the execution phase immediately without waiting for the outcome of the replication at ③ following the speculative replication approach. Between ③ and ⑤ processing acknowledgments messages are exchanged within the leader cluster instance. The replication layer ensures that replication requests are delivered reliably to the followers by ④ (i.e., using RECEIVEDATA API call). Replicated plans are exchanged in the follower clusters by ⑥. The replication layer responds to the leader set nodes by ⑦. After ⑦, leader nodes safely proceed with the commitment stage and commit transactions. At the follower clusters, the execution phase starts at ⑥, and the commitment stage starts at ⑧. The commitment stage at the follower nodes requires acknowledgments from nodes in their cluster instance to ensure that multi-partitioned transactions are processed successfully by all participating partitions (i.e., between ⑥ and ⑧). By ⑨ leader nodes respond to clients.

Discussion Note that the replication layer can respond to the replication request by the leader set of nodes before the followers receive the EQs. This invariant is stated as follows:

Invariant 1 (Replication Invariant)**.** *The leader nodes receive acknowledgments of their replication request from the replication layer if and only if the replication layer guarantees that followers eventually receive replicated data.*

It is the responsibility of the replication layer implementation to ensure the eventual delivery of replicated data. The above invariant allows some flexibility in implementing the replication layer, which can be a middleware-based or an integrated implementation.

Replication Payload Compression. Replication payload is a function of the batch size. Therefore for large batch sizes, they can be in the order of a few hundred kilobytes.

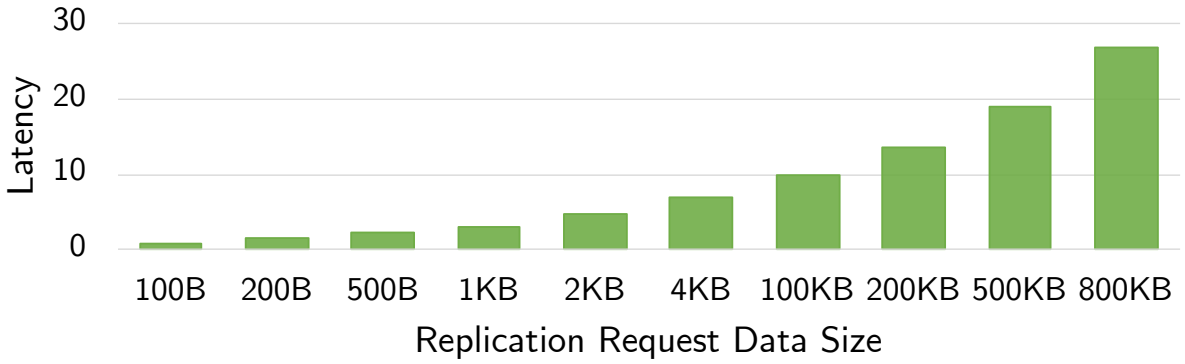


Figure 4.4. Zookeeper latency micro-benchmark. Latency is measured in milliseconds.

Algorithm 2 Execution algorithm

VARs: BM batch metadata

```
1: function executeBatch(tid)
2:     while not BM.done() do
3:         q ← BM.getTop(tid)
4:         while not q.empty() do
5:             f ← q.pop()
6:             executeTF(f)
7:             resolveDependency(f)
8:         end while
9:         if isRemote(q) then
10:            sendAck(q)
11:        else
12:            updateTC(q)
13:        end if
14:    end while
15: end function
```

At this scale of message sizes, the latency can be undesirably too high. Figure 4.4 shows the result of a micro-benchmark of submitting 100 concurrent requests to Zookeeper (as a replication service) while varying the payload size from 100 bytes to 800 kilobytes. Note that Zookeeper can only support a maximum of 1 megabytes of data stored as a single Zookeeper node. We can observe that for large message sizes, the latency can reach up to 26 milliseconds.

To reduce the payload size of replicated data in QR-Store, we compress replication data only using Snappy [83], we observed a reduction of payload sizes by 60%.

4.3.4 Speculative Execution Algorithm

In this section, we present the execution algorithm in QR-Store. The algorithm is simple, and its pseudo-code is presented in Algorithm 2. BM is a reference to the Batch metadata data structure, which is assumed to have global access. A worker thread at the execution phase keeps working on executing EQs until all EQs are processed. It retrieves the next available EQ at Line 3 using GETTOP. The returned EQ must satisfy the following conditions:

1. Condition 1: if a worker thread i gets EQ q using GETTOP, no other worker thread j gets q. The thread identifier *tid* is used to ensure this condition is satisfied.

2. Condition 2: The read/write sets of transaction fragments in q **do not** overlap with read/write sets of any other transaction fragment in EQs that remains in BM.

3. Condition 3: q has the highest priority in BM

These conditions ensure the following global execution priority invariant is maintained across all worker threads in the cluster when executing Line 6.

Invariant 2 (Global Execution Priority Invariant)**.** *Across all nodes in a cluster, transaction fragments from higher priority EQ* ***are always*** *executed before transaction fragments from lower priority EQs.*

After executing a transaction fragment, we need to resolve any data dependencies of that fragment (Line 7). An example of a data dependency is a transaction fragment that performs the following operation $f : x = x + y$. In this case, x and y are records that belong to different partitions. Reading record y is needed to resolve the dependency of computing the new value of x. Hence, to resolve the dependency on x, we need to send the value of y to the node executing the transaction fragment f, which is the node that holds record x. The transaction contexts maintain the state of transactions, and they are updated during execution. If q is a remote EQ, an ACK is sent to the original planning node (Line 10). When the ACK is received by the planning node, the transaction contexts of relevant transactions

are updated (e.g., updating the number of fragments that are completed). If q is local (i.e., q is planned by the same node that executed q), the transaction contexts are updated locally (Line 12).

This execution is speculative because transaction fragments from different transactions are executed and the commitment is done at a later stage. For example, an EQ q can contain transaction fragments from f_1 and f_2 from transactions t_1 and t_2, respectively. A worker thread executes f_1 followed by f_2. However, t_1 is committed later after executing f_2.

Discussion The main problem associated with speculative execution is the notion of cascading aborts [2]. DTP systems abort only if the transaction has explicit abort logic. For example, a transaction t_i that makes a product purchase would *abort if the product's stock* $== 0$. Any transaction that conflicts with t_i will also abort if it reads any records updated by t_i because values written by t_i are not committed and should not be visible. QR-Store keeps track of transaction conflict information in the form of a dependency graph. The graph is made available to the commitment stage so that transactions are committed according to the correct isolation level. We assume serializable isolation in throughout this chapter. However, as shown in [50], we can also support other isolation levels.

4.3.5 Commitment Algorithm

For transaction commitment, the original planner node act as the transaction coordinator for all transaction it planned. Thus, it requires receiving ACKs for all remote EQs. These ACK messages are communicated to the transaction coordinator node as remote EQs are executed. As an illustration, in Figure 4.3, they are communicated between (3) and (5) for the leader set, and between (6), and (8) for the replica sets.

Algorithm 3 shows the pseudo-code for the commitment algorithm used by the transaction coordinator nodes. Because a leader node can run multiple planning producing different sets of plans, each planning thread is identified by the *tid*. Thus, the *tid* is used to commit a transactions planned by a specific planner (Line 2). In Line 3, P is initialized to an empty FIFO queue to hold transactions pending commit. The order of commitment is concerned only with conflicting transactions. Non-conflicting transactions can commit in any order, and

Algorithm 3 Commitment Algorithm

VARs: BM batch metadata

```
1:  function commitBatch(tid)
2:      T ← BM.TC.getTransactions(tid)
3:      P ← empty FIFO queue for pending transactions
4:      for t ∈ T do
5:          status ← commitTxn(t)
6:          if not status then
7:              P.push(t)
8:          end if
9:      end for
10:     while not P.empty() do
11:         status ← commitTxn(P.head)
12:         if status then
13:             P.pop()
14:         end if
15:     end while
16: end function
```

our algorithm allows. In Lines 4 to 9, we perform a single iteration to commit transactions. In Line 5, the commitTxn function checks if the transaction can commit (i.e., all of its fragments are executed successfully). It returns *true* if the transaction t is committed, and *false* otherwise. If a transaction cannot commit at this time (Line 6), it is pushed into the pending transaction queue P for a later check, which happens in Lines 10 to 15. A transaction is checked at the head of the queue without removing it (Line 11). It is only removed if it is committed (Line 13).

Note that a single-threaded implementation of the commitment algorithm can join all transactions into a single set for commitment. The only requirement is that to ensure that the correct commit order of conflicting transactions is preserved.

Regardless of the implementation of the commitment algorithm, the commit stage needs to adhere to the following invariant. We use $po(t)$ to denote planning order, and $co(t)$ to denote the commitment order of transaction t, respectively.

Invariant 3 (Commitment Invariant)**.** *For any two conflicting transactions* t_i *and* t_j, $co(t_j) > co(t_i) \iff po(t_j) > po(t_i)$.

Based on the above three invariants, we state the following theorem and provide a proof sketch.

Theorem 4.3.1. *QR-Store's transaction processing protocol is safe and strictly serializable.*

Proof. It follows from the three invariants stated above that the QR-Store processes transactions with strict serializability. Invariant 3 ensures that the commitment order of conflicting transactions follows the planning order. Planning threads impose ordering by using the ordering property of queues in EQs. The order between EQs planned by different planning threads is determined by the priority order of the planning threads. Thus, there is a global partial order of all transaction fragments, which is preserved by Invariants 2 and 3. Furthermore, because the commitment stage does not start until the replication requests are acknowledged according to Invariant 1, the transaction commitment is safe. □

4.3.6 Latency Model

In this section, we model the latency for our queue-oriented transaction processing with replication. The key idea of performing speculative replication is to hide the replication latency overhead. The following equation models the latency of completing the processing of a single batch which is denoted as $\mathscr{T}_b$.

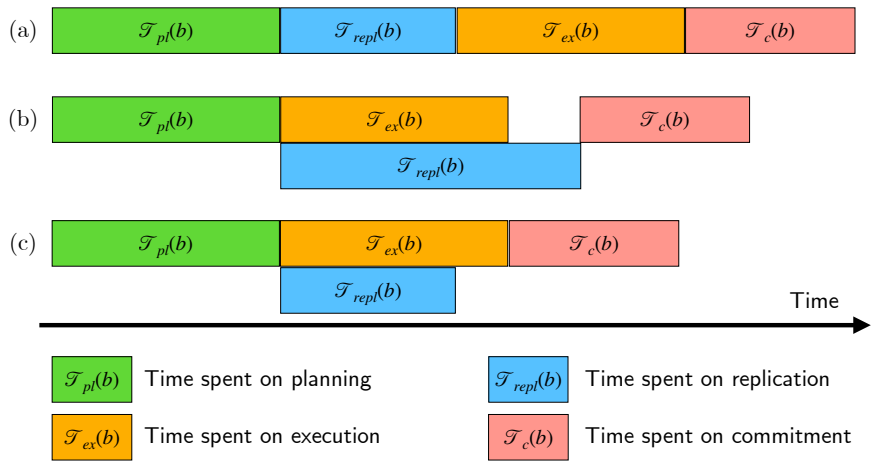


Figure 4.5. Illustrating Replication Overhead in QR-Store

$$\mathscr{T}(b) = \mathscr{T}_{pl}(b) + max(\mathscr{T}_{deliv.} + \mathscr{T}_{ex}(b), \mathscr{T}_{repl}(b)) + \mathscr{T}_{c}(b) \tag{4.1}$$

- $\mathscr{T}_{pl}(b)$ is the time spent in the planning phase for batch b
- $\mathscr{T}_{deliv.}$ is the time spent on delivering remote EQ messages for batch b
- $\mathscr{T}_{ex}$ is the time spent in the execution stage for batch b
- $\mathscr{T}_{repl}$ is the time spent waiting for the replication to be confirmed for batch b by the replication layer
- $\mathscr{T}_{c}$ is the time spent on committing transactions for batch b

In Figure 4.5, we show a visualization of three cases of queue-oriented replication. (a) depicts using synchronous replication in QR-Store. In this case, the replication request must be acknowledged before we start the execution phase. Thus, the overhead of replication is directly added to the latency of processing a batch in QR-Store. (b) and (c) in Figure 4.5 are the two cases of using the speculative replication approach. In (b), the replication takes longer than the execution, which forces the execution threads to wait for the replication confirmation before starting the commit stage. The optimal case is depicted by (c), which totally hides the replication latency, while in (b), the replication overhead is partially hidden.

4.3.7 Logging and Recovery in QR-Store

All proposed DTP systems assume a deterministic stored procedure transaction model (e.g., [2], [12], [14], [50], [76]). Furthermore, the stored procedure model assumes that the transaction logic is deterministic. In other words, the output is the same as long as the procedure is given the same input.

DTP systems use a combination of checkpointing and command-logging to facilitate logging and recovery. With command-logging, only the input of the transactions is logged, and on recovery, the log is applied from the latest stable checkpoint. Checkpointing can be done asynchronously to the disk to avoid blocking the transaction processing workflow.

In QR-Store, when a leader node crashes and resumes operation, the first step is to determine the new leader. The next step is to request all EQs from the current leader since the last stable checkpoint and execute them to recover the database partition state. After that, it acts as a follower by getting its plans from the replication layer.

Follower nodes detect leader nodes crashes via heartbeat messages exchanged periodically between leaders and followers. When followers detect that a leader has failed, they run a leader election process among them. Once a new leader is elected, the newly elected leader node requests leadership ACKs from other followers to determine the last committed batch. It requests any missing queues and replays them.

Using command-logging only with QR-Store introduces a recovery challenge. First, logged commands need to be planned again, which increases the latency of recovery. Second, when recovering multi-partition transactions, participation from all partitions is required to resolve data dependencies.

Therefore, instead of using command-logging and simply log transaction inputs, QR-Store create special write-only EQs that contain the last write operation of records accessed by planned EQs. These write-only EQs are logged to facilitate recovery. Thus, to recover a node's state, it is sufficient to request these write-only EQs from other nodes. This approach also resolves data dependencies associated with replaying multi-partition transactions because the logged value does not have any data dependencies.

4.4 Implementation

This section discusses the implementation aspects of the replication layer and optimizations related to synchronization granularity. We show the impact of these implementations and optimizations in Section 4.5.

4.4.1 Replication Layer Abstraction

As mentioned in Section 4.2 that we provide a simple API abstraction for the replication layer. We now give some details on the API, which consists of the following two functions.

replicateData(`meta_data, data, [callback]`). This is an asynchronous function that is called by the leader set of nodes (the dotted arrow in Figure 4.1 originating from the red circles). `meta_data` parameter can have some identifying information of the data being replicated (e.g., a batch identifier). The `data` parameter is a byte string of the data being replicated. `callback` parameter is a function that is called after the replicateData function completes. Since the function is asynchronous, the callback function provides a way to perform some actions (e.g., error handling).

receiveData(`meta_data, callback`). This is also an asynchronous function, and it is called by replicas to receive the replicated data. (the solid arrow in Figure 4.1 originating from the green replication layer) The `meta_data` parameter can have some identifying information about the replicated data from the replica's perspective. For example, it can include the expected batch identifier. The `callback` parameter is a function that is called with and passed the replicated data. It is used to construct the DTP system's representation of the replicated data from byte string passed to replicateData.

4.4.2 Replication Layer Implementations

Our prototype provides two implementations of the replication layer, and we describe these implementations in this section.

Middleware Replication The first one uses Zookeeper [77] as a middleware to implement the replication Layer. Leader servers and replica servers act as clients to the Zookeeper cluster. While leader servers make write requests, replica servers make read requests to get the replicated data. The Zookeeper cluster is a highly available system, and it does not constitute a single point of failure because it uses its internal replication and consensus protocols to ensure correct fail-over. The consensus protocol used by Zookeeper is called ZAB[84] The advantage of this approach is that it simplifies the replication layer shim implementations at the server nodes. The disadvantages include adding an overhead of the middleware to the transaction processing protocol. This approach is adopted by Calvin in their original proposal [12] and also in our experiments.

Integrated Replication The second implementation of the replication layer integrates a quorum-based replication protocol with the transaction processing protocol and effectively eliminates the middleware overhead. Our implementation is based on RAFT[82] and Viewstamped replication [81]. With this implementation, the leader server nodes send replicated data messages to replicas. On receiving replicated data messages, replicas reply with acknowledgment messages to leader servers. Depending on the number of node failures to tolerate, there is a minimum number of acknowledgment messages that confirm a successful replication. Let the number of node failures be denoted as f such that the total number replica for a given node is $n = 2f + 1$. The number of acknowledgment messages is $f + 1$.

4.4.3 Synchronization Granularity

In QR-Store, we support multiple granularities of transaction processing synchronization within a cluster. A coarse-grained synchronization puts node-level barriers between batches. Hence, before any thread can start processing the next batch, it has to wait for all other nodes to finish processing the current batch. At the cost of additional implementation complexity, it is possible to have a more fine-grained synchronization at the thread level. This thread-level synchronization allows worker threads to start the planning phase of the next batch as soon as other nodes acknowledge the execution of their respective planned EQs. Thread-level synchronization improves the concurrency of the phases across batches. Our previous prototype implementation for Q-Store [50] uses node-level synchronization while our current prototype uses thread-level.

4.5 Evaluation

In this section, we present our experimental evaluation. We implement the three case studies that we discussed in Section 4.2. We use the first replication point for Calvin, and use the second replication point for QR-Store and a fully replicated version of QueCC (denoted as QueCC-R). The experimental study of the other replication points admitted by our proposed framework in Section 4.2 remains future work.

Table 4.1. System and workload configuration parameters. Default values are in parenthesis. Default values are used unless stated otherwise.

P#	**Parameter Name**	**Possible Parameter Values**
$P1$	% of multi-partition txns.	$0\%, 10\%, 15\%, (50\%), 75\%, 100\%$
$P2$	Zipfian's theta	$(0.0), 0.4, 0.6, 0.8, 0.9, 0.99$
$P3$	Operations/txn.	$2, 4, 8, 12, (16)$
$P4$	Batch sizes	$2K, 5K, 10K, (20K), 40K, 80K, 100K$
$P5$	Server nodes counts	$2, 4, 8, (16)$
$P6$	Replication factor	$(0), (1), (2), 4, 6, 8$

We mainly focus on QR-Store with various replication factors configured. The current prototype of QR-Store is the optimized and improved version of our previous work presented in [50]. In our comparison with Calvin, we use Zookeeper as the implementation of the replication layer as it is originally presented in [12].

4.5.1 Experimental Setup

We use up to 64 `c2-standard-8` instances on Google Cloud Platform to run our experiments. These instances have 8 vCPUs, 32GiB of RAM, and the default egress network bandwidth available is 16Gbps. Each node runs Ubuntu 18.04 (bionic beaver), and the codebase is compiled with the $-O3$ compiler optimization flag. We use four worker threads and four communication threads. Threads are pinned to cores to minimize potential variance due to the operating system.

Furthermore, our Calvin's implementation dedicates one worker thread for the sequencer role and another worker thread for the scheduler role. This configuration leaves two threads for processing transactions [3].

Each data point is the average of three trials. Each trial consists of a warm-up phase of 60 seconds where measurements are not collected, followed by a measurement phase of 60 seconds.

System and workload parameters In Table 4.1, we present all system and workload parameters used in our experiments. $P1$ is the percentage of multi-partition transactions (MPTs) in the workload. An MPT accesses more than one partition and requires a dis-

[3]↑using additional worker threads reduces Calvin's performance by nearly 50% according to our observations

tributed commit protocol. $P2$ is the parameter that controls the access distribution of transactions. Higher values of θ make the access skewed to a small set of records. $P3$ is the parameter that controls the number of operations per transaction and requires transaction atomicity when there is more than one operation in a transaction. A transaction with a single operation is atomic by definition. $P4$ is the size of transaction batches that are processed by queue-oriented transaction processing systems such as QR-Store, Q-Store and QueCC. $P5$ controls the number of server nodes used in a cluster. A cluster of nodes forms an instance of the database, and each node manages a single partition. By default, we use one client node per server node. $P6$ represents the number of replicas used per cluster of servers. For example, a value of 2 means that there are 2 additional replicated database instances.

4.5.2 Experimental Results

We first study the impact of replication. We use three configurations. As a baseline, we use Q-Store which does not perform replication. QR-Store-rf1 and QR-Store-rf2 has a replication factor of 1 and 2, respectively.

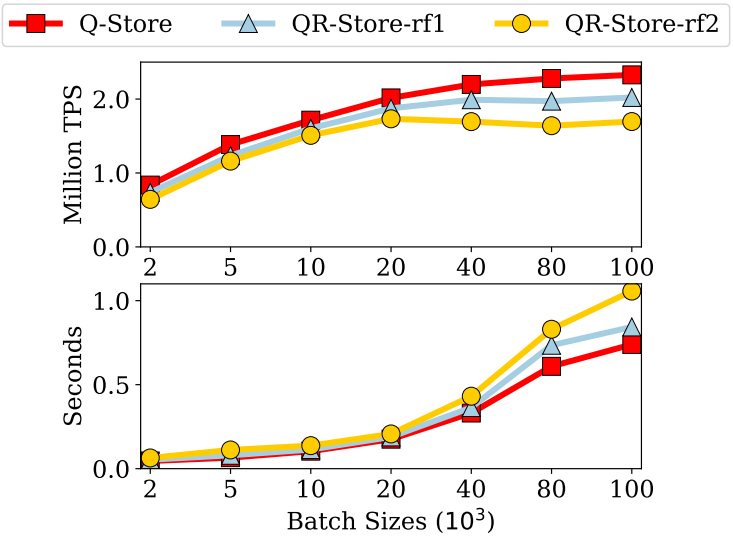


Figure 4.6. Varying Batch Size

Varying the batch size In this set of experiments, we want to understand the effect of the batch size on the performance of QR-Store. We use 50% MPT transactions in the

workload, uniform access distribution (i.e., a value of $\theta = 0.0$), 50% update operation per transaction, 16 operation per transaction, and each transaction access 8 partitions. The number of server nodes is 16.

Figure 4.6, shows the system throughput and the 99^{th} percentile latency of transaction processing. Q-Store which is the configuration without replication performs the best both in terms of throughput and latency (up to 2.3 million TPS executed in under 750 milliseconds). While Q-Store's throughput performance keeps increasing with batch sizes greater than $20K$ transactions, the latency also increases significantly. With larger batches, worker threads spend more time planning and executing transactions. Also, the size of messages exchanged between the server nodes within the cluster becomes larger. Moreover, beyond the $20K$ batch size, the gap in performance Q-Store, and the replicated configurations (i.e., QR-Store-rf1 and QR-Store-rf2) becomes more significant because the leader cluster needs to prepare and replicate larger plans. With QR-Store-rf2 the number of messages that are sent by the leader nodes is twice the number sent by the QR-Store-rf1 configuration. Hence, as these messages become larger, the computation and communication requirements increase. For example, for $rf = 2$ the 99^{th} percentile latency increases from 30% at $40K$ batches to 43% at $100K$. Notably, at $20K$ batches, the latency overhead is only 16% in this workload configuration.

Scalability when increasing number of server nodes In this set of experiments, the percentage of MPT is 50%, the Zipfian theta parameter is set to $\theta = 0.0$, the percentage of write operations is 50%, the number of operations per transaction is 16, the batch size is set to $20K$ transactions, and we force each transaction to access all available partitions. In other words, MPTs will always access all servers. We vary the number of server nodes from 2 to 16.

Figure 4.7 shows that all configurations scale linearly as we add more nodes into the server cluster. The linear scaling is because operations in each transaction are processed in parallel by all available nodes. Notably, the throughput performance reaches 2, 1.8, and 1.7 million transactions per second for Q-Store, QR-Store-rf1, and QR-Store-rf2, respectively. The 99^{th} percentile latency remains under 216 milliseconds.

Figure 4.7. Scalability with increasing the number of servers/partitions in a cluster instance.

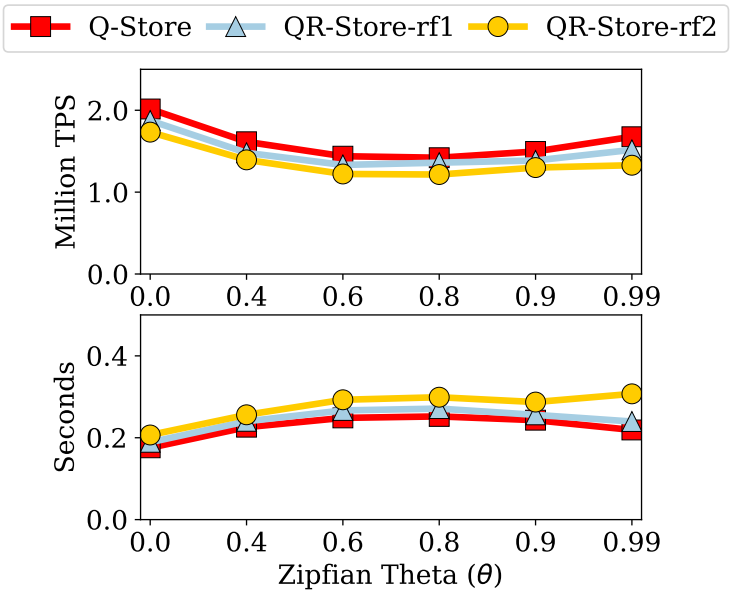


Figure 4.8. Varying Data Access Contention

When using higher replication factors (i.e., QR-Store-rf1, and QR-Store-rf2), the impact of replication becomes larger. In our experiments with 16 server nodes, the performance drops by 15%. We believe that this is a reasonable cost to ensure fault tolerance.

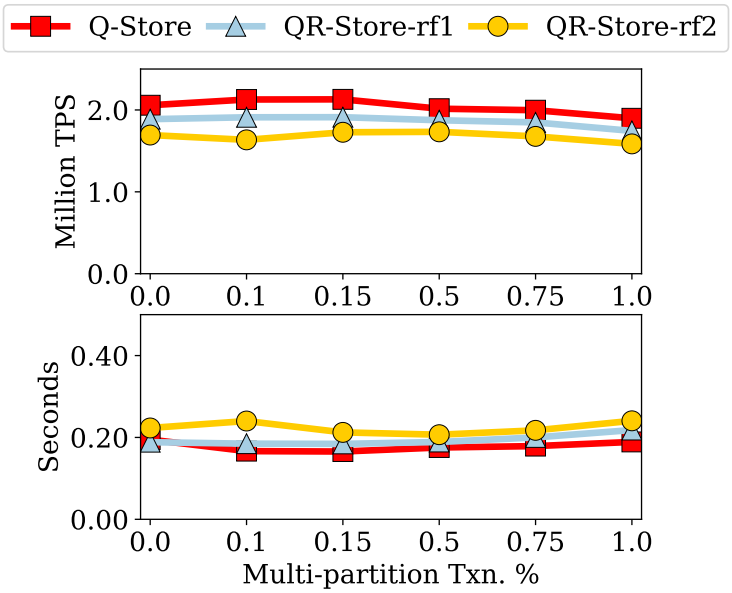


Figure 4.9. Varying Multi-partition transaction rate

Varying Data Access Contention In Figure 4.8, we vary the value of the Zipfian distribution θ from 0.0 to 0.99. Using 0.0 for θ creates uniform access to database records, while a value 0.99 creates extremely skewed access resulting in an increased contention on database records.

Our queue-oriented approach naturalizes the high contention because the operations accessing the same set of records are placed in the same EQ and are executed by the same worker thread. However, as shown in Figure 4.8, we observe a decrease in performance when there is medium contention (i.e., $0.4 - 0.8$). For Q-Store, the throughput drops by $19 - 29\%$, and the latency increases by $28 - 39\%$. The throughput drops by $19 - 26\%$ and $17 - 26\%$, and the latency increases by $23 - 42\%$ and $26 - 42\%$ for QR-Store-rf1 and QR-Store-rf2, respectively. At medium contention, some EQs contain more operations than others, which increases the execution time. However, at high contention (i.e., $0.9 - 0.99$), the performance gets better because the caching becomes more effective as most operations in the large queues access a small set of records. Notably, at low contention (i.e., $\theta = 0.0$), the overhead of replication is 6% and 13% for QR-Store-rf1 and QR-Store-rf2, respectively.

Varying MPT percentage Now, we look into the effect of increasing the percentage of the multi-partition transactions in the workload. For this set of experiments, we use uniform access and enforce each MPT to access 8 partitions of the 16 partitions. The update operation percentage remains at 50%. Increasing the MPT percentage increases the sizes of remote EQs that are planned. The throughput performance gets better at a low ratio of MPT in the workload by 6.5%, 10.5% and 24% for Q-Store, QR-Store-rf1, and QR-Store-rf2, respectively, because some operations are executed remotely by other nodes which reduced the load on local worker threads. However, as we increase the ratio, the performance starts dropping to even below the performance of a pure single partition workload because remote executions take longer times to be acknowledged.

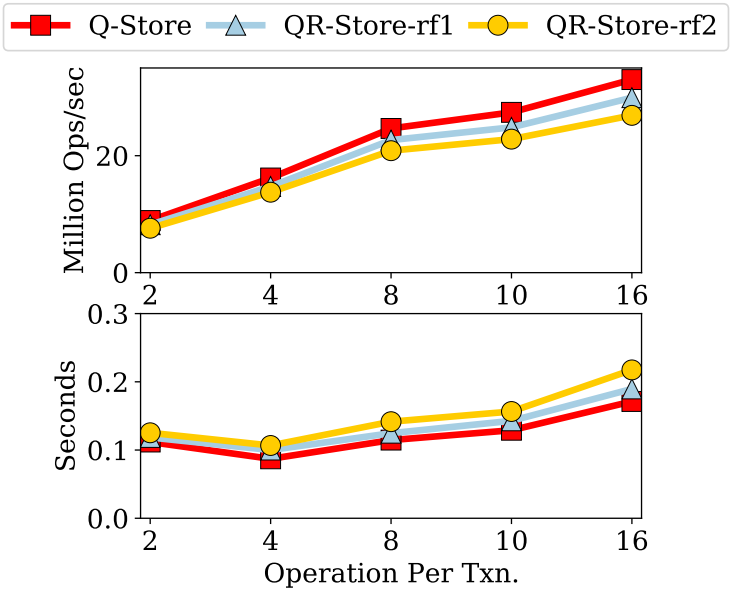


Figure 4.10. Varying operation per transaction

Varying the number of operations per transaction The number of operations per transaction represents the transaction size. Again, we fix the other system and workload parameters to their default values and vary the number of operations per transaction. In Figure 4.10, we use the number of operations processed per second instead of the number of operations. All configurations scale their throughput performance linearly as we increase the number of operations, and the throughput performance reaches up to 33, 30, and 27 mil-

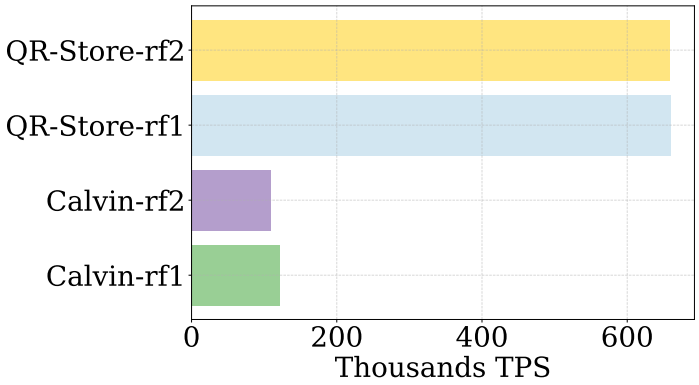


Figure 4.11. Comparison with Calvin

lion operations per second for Q-Store, QR-Store-rf1 and QR-Store-rf2, respectively. The 99^{th} percentile latency is between $110 - 220$ milliseconds. With a low number of operations per transaction, the communication overhead is more significant. As the number of operations per transaction increases, the EQs become larger and more efficient to execute. With replication the throughput performance drops by 9% and 18% for QR-Store-rf1 and QR-Store-rf2, respectively.

Comparison with Calvin

In this set of experiments, we want compare QR-Store's performance to Calvin's [12]. We implemented Calvin's approach to replication which uses Paxos via Zookeeper. QR-Store's approach uses an integrated replication protocol (Section 4.4). Hence, QR-Store's implementation of the replication layer eliminates the overhead of a replication middleware. We use four server nodes per cluster and enable compression for Calvin replicated messages. We use a highly skewed workload for the workloads with $\theta = 0.9$, 10 operations per transaction, 50% MPT, update ratios, and force each transaction to access two partitions.

As shown in Figure 4.11, QR-Store's configurations (denoted as QR-Store-rf1 and QR-Store-rf2) outperforms Calvin's configurations (denoted as Calvin-rf1 and Calvin-rf2) by up to 6×. Multiple factors are contributing to this improvement. The first one is the use of the queue-oriented speculative transaction processing model, which is more efficient than Calvin's transaction execution model. The second factor is the use of the integrated replication implemen-

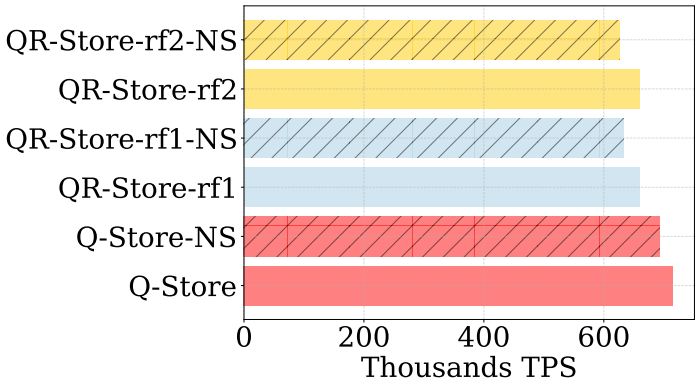


Figure 4.12. Node granularity vs. fine granularity synchronization

tation as opposed to Zookeeper, which introduces significant replication processing overhead. Using the integrated approach, QR-Store introduced no more than 8% performance overhead compared to the Q-Store within a four-node cluster configuration.

Impact of node-granularity synchronization One of the key optimizations that we introduced in our current prototype is the granularity of synchronization in QR-Store. In our previous work [50], we adopted a node-level synchronization protocol that runs after processing a batch, which synchronizes all worker threads before they start working on the next batch. Using this approach simplified our prototype implementation and allowed us to avoid locking shared data structures. However, it also introduced unnecessary idle time periods where worker threads can perform useful work for the next batch.

In our current prototype, we designed and implemented a fine-grained synchronization protocol that increases the concurrency of planning and execution phases. With our queue-oriented transaction processing paradigm, a node-level partition is further partitioned by planning threads. Instead of waiting for every other node in the cluster before starting its planning phase, it starts the planning phase, and it only waits for ACK messages for the remote EQs that it planned before delivering the EQs for the new batch. Hence, this approach effectively implements a fine-grained synchronization protocol at the thread level instead of the node level.

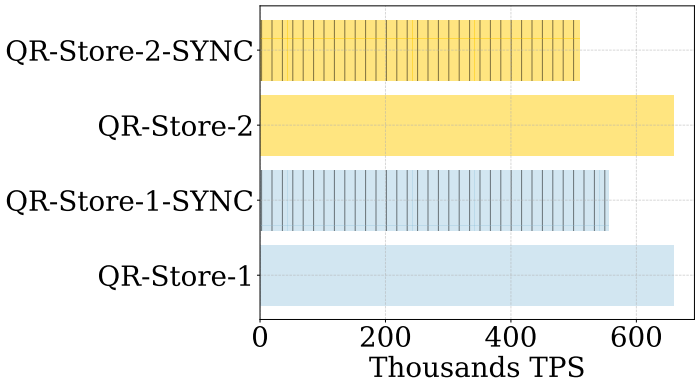


Figure 4.13. Speculative replication versus synchronous replication

Figure 4.12, shows a comparison of node-level synchronization and thread-level synchronization with various configurations of QR-Store. We use four server nodes per cluster and the default workload parameters. The configurations that use node level synchronization are denoted with -NS suffix. The thread-level synchronization technique provides up to 5% improvement in throughput performance and up to 14% in latency reduction.

Speculative replication vs. synchronous Another key technique in QR-Store is the concept of speculative replication. The basic idea is that instead of waiting for the replication to complete before starting the execution phase of a batch, QR-Store speculates that the replication is expected to succeed and starts the execution phase without waiting. However, before starting the commit stage, the system waits for *acknowledgments* confirming the success of the replication requests. In Figure 4.13, we show a comparison between the two techniques (the synchronous configuration is denoted with a -SYNC suffix). The speculative replication technique improves the performance by up to 30% with a four-node cluster and the default workload parameters.

Impact of the replication factor The replication factor dictates the number of replicas of the database instance. The leader set of server nodes perform a proportional amount of work to the number of configured replicas. We perform a set of experiments involving four nodes per cluster and a fully replicated configuration. The fully replicated configuration implements the third case study described in Section 4.2.3. In Figure 4.14, QueCC-R is a

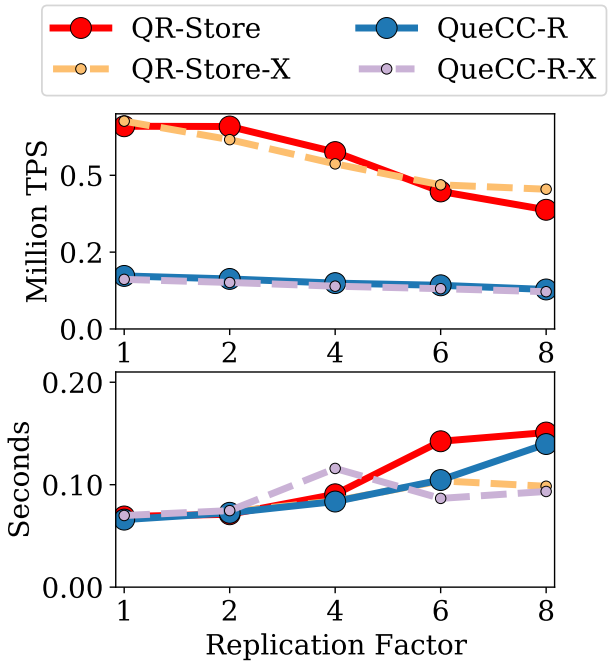


Figure 4.14. Impact of the replication factor and replication compression. Dashed lines represent configurations with replication compression enabled. Four nodes per cluster for QR-Store

non-partitioned (i.e., single-node database instance) and replicated configuration of QR-Store while QR-Store is the partitioned and replicated configuration with four-node per cluster. We increase the replication factor from 1 up to 8. As we can see, the overhead of replication beyond a replication factor of four becomes significant. It reduces the performance by up to 41% and 26% as we increase the replication factor to 8 for QR-Store and QueCC-R, respectively. The large drop is due to the increased demand for network resources as the number of replication messages increases proportionally to the replication factor.

Impact of using compression for replication Intuitively, using compression reduces the number of bytes that go over the network for replication messages by the leader set of nodes in both QR-Store and QueCC-R. However, it increases the CPU computation requirements on the leader set of nodes as more CPU cycles are needed to perform the compression. Hence, compression is not a silver bullet and is not always beneficial. We conduct experiments to demonstrate that. Figure 4.14 shows that compression can be beneficial when the replication factor is high (e.g., at 6 or 8). The 99^{th} percentile latency improves by up to

35%. Compression is also beneficial for Calvin because it uses Zookpeer as the replication layer. In our experiments, Calvin latency improved by up to 53%. This result agrees with our micro-benchmark result shown in Figure 4.4. However, it increases the work on the leader nodes in all other cases, which negates its benefits.

4.6 Related Work

In this section, we discuss relevant work from the literature. In this chapter, we addressed an important challenge of high-performance replication in distributed deterministic transaction processing systems. However, the database replication techniques have been studied since several decades ago [85]. These techniques are mainly studied with respect to two dimensions. The first dimension is whether to allow transactions to update at any replica or designate a primary copy replica. The second dimension is when to synchronize replicas and whether we should do that synchronization eagerly or lazily. Furthermore, traditional database replication techniques reuse non-deterministic transaction processing protocols e.g., 2PL, 2PC [1] and OCC [48]. The reader is referred to existing literature that cover the traditional database replication techniques very well (e.g., [18], [72]–[74], [86]) for more details. Compared to traditional database replication techniques, the techniques proposed in this chapter are deterministic, speculative, and adopt the queue-oriented transaction processing paradigm [76]. Hence, this chapter explores a new research territory.

Replication Frameworks Wiesmann et al. [87] proposed a general replication framework to study replication techniques developed by the database systems research community and the distributed systems research community. However, it does not address the design choices made by DTP systems. Our proposed general framework complements their replication framework by focusing on replications in DTP systems.

Deterministic transaction processing protocols Deterministic transaction processing approaches are shown to process transactions more efficiently when compared to non-deterministic approaches. Recently, there have been many proposals for deterministic transaction processing protocols. These systems can be *centralized* (e.g., [2], [10], [15], [16], [26]) or *distributed* (e.g., [12], [14], [22], [50]). The main focus of these proposals is on the con-

currency control aspects of processing transactions. In contrast, this chapter goes beyond existing deterministic transaction processing systems by addressing the replication challenge in a systematic way and build on queue-oriented principles, which allows efficient transaction processing and database replication.

Replication and Consensus Protocols In the distributed systems community, consensus and replication have received a great deal of attention (e.g., [80]–[82], [88]). Such work focused on state-machine replication and aimed to achieve linearizability. In contrast, this chapter is concerned with strict serializability guarantees of transaction processing on distributed, partitioned, and replicated databases.

4.7 Conclusion

In this chapter, we propose a generalized framework for designing replication schemes for distributed DTP systems. Using the framework, we study three cases from the literature and discuss how replication can be reasoned about. We propose a novel queue-oriented speculative replication technique and describe how it is supported in QR-Store. Finally, we perform an extensive evaluation of several configurations of QR-Store and demonstrate efficient replicated transaction processing that can reach up to 1.9 million replicated transactions per second in under 200 milliseconds and a replication overhead of $8\% - 25\%$ compared to non-replicated configurations.

5. CONCLUSIONS

In this chapter, we provide concluding remarks on the contributions of this dissertation. First, a brief summary of this dissertation is provided. Moreover, the current limitations of queue-oriented transaction processing are discussed. Finally, recommendations on the future research directions are discussed.

5.1 Summary

In this dissertation, we addressed challenges in deterministic transaction processing in main-memory databases. The first challenge is concerned with how to efficiently utilize the high number of cores available in modern computing infrastructure. The second challenge is concerned with how to perform distributed transaction processing efficiently. Finally, the third challenge is concerned with how to perform replication efficiently in deterministic transaction processing systems to support fault tolerance. Our focus in this dissertation is on developing queue-oriented principles for efficient transaction processing. We prototype the proposed techniques and extensively evaluate them using standard transaction processing benchmarks. Our experimental result shows significant improvement in the transaction processing system performance. However, our proposed queue-oriented transaction processing approach has several limitations, and these limitations are discussed in the following sections.

5.2 Current Limitations of Queue-oriented Transaction Processing

Advantages and disadvantages of deterministic transaction processing are discussed in the literature [21]. Since our queue-oriented transaction processing is deterministic, these advantages and disadvantages extend to our proposed queue-oriented paradigm. Previous chapters demonstrated that it is possible to achieve highly efficient transaction processing performance by adopting queue-oriented transaction processing principles for designing and implementing distributed and replicated deterministic transaction processing systems. Like other deterministic transaction processing systems [12], [63], queue-oriented transaction pro-

cessing assumes the knowledge of the full read/write set of transactions prior to the planning phase. In this section, we discuss existing techniques that are concerned with this assumption. Furthermore, we discuss an intra-transaction property that inhibits the performance in queue-oriented transaction processing.

5.2.1 Knowledge of the Read/Write Sets

A key limitation of the deterministic transaction processing is that the knowledge of the full read/write sets is required. However, there are few approaches to address this problem. One approach is to run the transaction without committing its write-set to compute the full read/write sets [12], [89]. In general, this approach does not guarantee the finality of the read/write set when running the transaction. Another approach is to partially execute the transaction over multiple batches instead of a single batch.

The above approaches aim to do the heavy lifting on behalf of database application developers. Conversely, it is possible to expose an API to the application developers and specify the read-write sets in the transaction specification. While this approach puts a larger burden on application developers compared to earlier approaches, we believe that it is a very reasonable trade-off because it is typical for application developers to optimize their applications for performance, and this API can be used as an optimization tool.

5.2.2 Resolving Data Dependencies

Our proposed queue-oriented execution exploits any available intra-transaction parallelism. However, intra-transaction data dependencies among transaction fragments reduce the parallel execution efficiency by forcing the execution of an EQ to be stalled and cause execution threads to switch to other available queues. One way to remedy this is by performing out-of-order queue execution, which can minimize the switching overhead. However, the gained execution efficiency of this approach heavily depends on the workload.

Moreover, when executing multi-partition transactions issue of data dependencies becomes even more prominent as resolving data dependencies require a trip over network which more costly than resolving a local dependency. Fortunately, our planning phase can lever-

aged to reduce cost. To illustrate this benefit, consider two transactions $t_1 : \{r_1(a), r_1(c), a = a + c, w_1(a), Commit\}$ and $t_2 : \{r_2(a), r_2(c), b = b + c, w_2(b), Commit\}$. t_1 increments the value of record a by the value of c. t_2 performs a similar operation with record b. In this example, records a and b reside on the same partition which record c resides on a remote partition in a different node. Transaction fragments $w_1(a)$ and $w_2(b)$ To complete the write operation $w_1(a)$, have data dependencies on $r_1(c)$ and $r_2(c)$, respectively. To resolve these data dependencies, $r_1(c)$ and $r_2(c)$ must be executed and the value of record c is propagated to the node executing $w_1(a)$ and $w_2(b)$. Because the both transaction t_1 and t_2 are planned by the same planning thread, the planning thread can indicate that once the dependency on $w_1(a)$ is resolved, the same value of record c can be used to resolve the dependency on $w_2(b)$. In the next section, we discuss future research directions.

5.3 Future Research Directions

In this section, we discuss three future research directions. The first one is to support well-known features of relational database management systems in our proposed queue-oriented transaction processing paradigm. The second direction is applying queue-oriented principles to process transactions in blockchain systems. The final one is to apply machine learning techniques to build self-learning and adaptive queue-oriented transaction processing systems.

5.3.1 Challenges in the queue-oriented transaction processing paradigm

Relational database systems had come a long way since their inception decades ago. In this dissertation, we only scratched the surface of the queue-oriented paradigm. Our current prototypes use hash-based indexes to look up records for point-based queries. Supporting secondary indexes, triggers, and range-based queries remain open problems within our proposed paradigm. Furthermore, our transactions are written in C++. Adding support for more standard database application programming languages such as PL/SQL can lead to interesting engineering challenges.

From a storage point of view, we currently use a single version storage system to maintain a single version per record. It remains interesting to explore multi-versioned storage and time-traveling queries.

In Chapter 3, we outlined how we support read-committed isolation in addition to the serializable isolation. Supporting other isolation levels such as snapshot isolation is also an interesting challenge to address.

Moreover, worker threads in our system prototypes alternate between the planning phase and the execution phase. We plan to explore pipelined architectures where the worker threads work concurrently and collaboratively on the planning and execution phases.

Furthermore, we currently use a simple round-robin to assign a transaction to planning threads as input to the planning phase. Thus, there is no notion of a user-defined priority level. Supporting user-defined priorities for transactions while maintaining a certain level of fairness is also an interesting future direction.

Finally, in our current prototypes, we use simple planning techniques to avoid adding significant overhead to the planning phase. For instance, we use a range-based partitioning approach of record identifiers and map operations to these ranges to form the execution queues. However, our simple techniques may not produce optimal plans. As mentioned earlier, dependencies between transaction fragments in different queues can reduce the parallel execution efficiency. Hence, developing planning techniques that can produce near-optimal plans that minimize both dependencies among queues and the planning overhead is yet another interesting research direction.

5.3.2 Applications of the queue-oriented paradigm

Future work includes exploring the application of our queue-oriented transaction processing principles to design and implement efficient blockchain transactions. There are renewed research interests in byzantine fault-tolerance for transaction processing [24], [90]–[95]. We plan to support Byzantine fault tolerance for database transactions using our proposed queue-oriented transaction paradigm. On the one hand, blockchain transactions are deterministic, which aligns with the kind of transactions that our paradigm supports. On the

other hand, it is very challenging to design and implement efficient, Byzantine fault-tolerant protocols. We believe that the design principles behind QR-Store can lead to efficient Byzantine fault-tolerant protocols, as very few blockchain proposals look at optimizing execution.

Another interesting research direction is to apply queue-oriented transaction processing principles to build location-based services, spatial, spatiotemporal databases and data streaming systems (e.g., [96]–[98]). By modeling operations in these systems as transactions, the queue-oriented transaction processing techniques have the potential to improve the performance of these systems.

5.3.3 Using ML for adapting queue-oriented transaction processing systems

As with many complex software systems, queue-oriented transaction processing systems have many system parameters that can be tuned to optimize the system performance observed by the end-user. Hence, it is interesting to explore how to extend these systems and give them the ability to adapt to workload changes or other user requirements in a cloud environment dynamically by leveraging advanced techniques from the machine learning research literature.

REFERENCES

[1] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992, ISBN: 1-55860-190-2. [Online]. Available: http://portal.acm.org/citation.cfm?id=573304.

[2] T. M. Qadah and M. Sadoghi, "QueCC: A Queue-oriented, Control-free Concurrency Architecture," in *Proceedings of the 19th International Middleware Conference*, (Rennes, France), ser. Middleware '18, New York, NY, USA: ACM, 2018, pp. 13–25, ISBN: 978-1-4503-5702-9. DOI: 10.1145/3274808.3274810. [Online]. Available: http://doi.acm.org/10.1145/3274808.3274810.

[3] Hewlett Packard Enterprise, "HPE superdome servers," 2017. [Online]. Available: https://www.hpe.com/us/en/servers/superdome.html.

[4] Sgi, "SGI UV 3000 and SGI UV 30," 2017. [Online]. Available:
.

[5] Mellanox Technologies, "Multicore processors overview," 2017. [Online]. Available:
.

[6] Hewlett Packard Labs, "The Machine: A new kind of computer," 2017. [Online]. Available: http://labs.hpe.com/research/themachine.

[7] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker, "Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores," *Proc. VLDB Endow.*, vol. 8, no. 3, pp. 209–220, Nov. 2014, ISSN: 2150-8097. DOI: 10.14778/2735508.2735511. [Online]. Available: http://dx.doi.org/10.14778/2735508.2735511.

[8] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden, "Speedy Transactions in Multicore In-memory Databases," in *SOSP*, ACM, 2013, pp. 18–32, ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522713. [Online]. Available: http://dx.doi.org/10.1145/2517349.2522713.

[9] H. Lim, M. Kaminsky, and D. G. Andersen, "Cicada: Dependably Fast Multi-Core In-Memory Transactions," in *Proc. SIGMOD*, ACM, 2017, pp. 21–35, ISBN: 978-1-4503-4197-4. DOI: 10.1145/3035918.3064015. [Online]. Available: http://dx.doi.org/10.1145/3035918.3064015.

[10] C. Yao, D. Agrawal, G. Chen, Q. Lin, B. C. Ooi, W. F. Wong, and M. Zhang, "Exploiting Single-Threaded Model in Multi-Core In-Memory Systems," *IEEE TKDE*, vol. 28, no. 10, pp. 2635–2650, 2016. DOI: 10.1109/TKDE.2016.2578319. [Online]. Available: http://dx.doi.org/10.1109/TKDE.2016.2578319.

[11] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki, "Data-oriented Transaction Execution," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 928–939, Sep. 2010, ISSN: 2150-8097. DOI: 10.14778/1920841.1920959. [Online]. Available: http://dx.doi.org/10.14778/1920841.1920959.

[12] A. Thomson, T. Diamond, S. C. Weng, K. Ren, P. Shao, and D. J. Abadi, "Calvin: Fast Distributed Transactions for Partitioned Database Systems," in *Proc. SIGMOD*, ACM, 2012, pp. 1–12, ISBN: 978-1-4503-1247-9. DOI: 10.1145/2213836.2213838. [Online]. Available: http://dx.doi.org/10.1145/2213836.2213838.

[13] R. Harding, D. Van Aken, A. Pavlo, and M. Stonebraker, "An Evaluation of Distributed Concurrency Control," *Proc. VLDB Endow.*, vol. 10, no. 5, pp. 553–564, Jan. 2017, ISSN: 2150-8097. DOI: 10.14778/3055540.3055548. [Online]. Available: http://dx.doi.org/10.14778/3055540.3055548.

[14] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi, "H-store: A High-performance, Distributed Main Memory Transaction Processing System," *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1496–1499, Aug. 2008, ISSN: 2150-8097. DOI: 10.14778/1454159.1454211. [Online]. Available: http://dx.doi.org/10.14778/1454159.1454211.

[15] J. M. Faleiro, A. Thomson, and D. J. Abadi, "Lazy Evaluation of Transactions in Database Systems," in *Proc. SIGMOD*, ACM, 2014, pp. 15–26, ISBN: 978-1-4503-2376-5. DOI: 10.1145/2588555.2610529. [Online]. Available: http://dx.doi.org/10.1145/2588555.2610529.

[16] J. M. Faleiro and D. J. Abadi, "Rethinking Serializable Multiversion Concurrency Control," *Proc. VLDB Endow.*, vol. 8, no. 11, pp. 1190–1201, Jul. 2015, ISSN: 2150-8097. DOI: 10.14778/2809974.2809981. [Online]. Available: http://dx.doi.org/10.14778/2809974.2809981.

[17] K. Ren, J. M. Faleiro, and D. J. Abadi, "Design Principles for Scaling Multi-core OLTP Under High Contention," in *Proc. SIGMOD*, ACM, 2016, pp. 1583–1598, ISBN: 978-1-4503-3531-7. DOI: 10.1145/2882903.2882958. [Online]. Available: http://dx.doi.org/10.1145/2882903.2882958.

[18] B. Kemme and G. Alonso, "Don'T Be Lazy, Be Consistent: Postgres-R, A New Way to Implement Database Replication," in *Proc. VLDB*, Morgan Kaufmann Publishers Inc., 2000, pp. 134–143. [Online]. Available: http://dl.acm.org/citation.cfm?id=645926.671855.

[19] R. Jimenez-Peris, M. Patino-Martinez, and S. Arevalo, "Deterministic scheduling for transactional multithreaded replicas," in *Proc. IEEE SRDS*, 2000, pp. 164–173. DOI: 10.1109/RELDI.2000.885404. [Online]. Available: http://dx.doi.org/10.1109/RELDI.2000.885404.

[20] A. T. Whitney, D. Shasha, and S. Apter, "High Volume Transaction Processing without Concurrency Control, Two Phase Commit, Sql or C++," in *HPTS*, 1997.

[21] K. Ren, A. Thomson, and D. J. Abadi, "An Evaluation of the Advantages and Disadvantages of Deterministic Database Systems," *Proc. VLDB Endow.*, vol. 7, no. 10, pp. 821–832, Jun. 2014. DOI: 10.14778/2732951.2732955. [Online]. Available: http://dx.doi.org/10.14778/2732951.2732955.

[22] Y. Lu, X. Yu, L. Cao, and S. Madden, "Aria: A fast and practical deterministic OLTP database," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 2047–2060, Jul. 1, 2020, ISSN: 2150-8097. DOI: 10.14778/3407790.3407808. [Online]. Available: https://doi.org/10.14778/3407790.3407808.

[23] S. Gupta and M. Sadoghi, "EasyCommit: A Non-blocking Two-phase Commit Protocol," in *EDBT*, 2018. DOI: 10.5441/002/edbt.2018.15.

[24] S. Gupta and M. Sadoghi, "Blockchain Transaction Processing," in *Encyclopedia of Big Data Technologies*, S. Sakr and A. Zomaya, Eds., Cham: Springer International Publishing, 2018, pp. 1–11, ISBN: 978-3-319-63962-8. DOI: . [Online]. Available: .

[25] A. Adya, B. Liskov, and P. O. Neil, "Generalized isolation level definitions," in *Proc. ICDE*, vol. 0, IEEE, 2000, pp. 67–78, ISBN: 0-7695-0506-6. DOI: 10.1109/icde.2000.839388. [Online]. Available: http://dx.doi.org/10.1109/icde.2000.839388.

[26] J. M. Faleiro, D. J. Abadi, and J. M. Hellerstein, "High Performance Transactions via Early Write Visibility," *Proc. VLDB Endow.*, vol. 10, no. 5, pp. 613–624, Jan. 2017, ISSN: 2150-8097. DOI: 10.14778/3055540.3055553. [Online]. Available: http://dx.doi.org/10.14778/3055540.3055553.

[27] D. Shasha, F. Llirbat, E. Simon, and P. Valduriez, "Transaction Chopping: Algorithms and Performance Studies," *ACM Trans. Database Syst.*, vol. 20, no. 3, pp. 325–363, Sep. 1995, ISSN: 0362-5915. DOI: 10.1145/211414.211427. [Online]. Available: http://dx.doi.org/10.1145/211414.211427.

[28] 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, New York, NY, USA: ACM, 2016, pp. 1643–1658, ISBN: 978-1-4503-3531-7. DOI: 10.1145/2882903.2882934. [Online]. Available: http://doi.acm.org/10.1145/2882903.2882934.

[29] P. A. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling, "High-performance Concurrency Control Mechanisms for Main-memory Databases," *Proc. VLDB Endow.*, vol. 5, no. 4, pp. 298–309, Dec. 2011, ISSN: 2150-8097. DOI: 10.14778/2095686.2095689. [Online]. Available: http://dx.doi.org/10.14778/2095686.2095689.

[30] M. Sadoghi, S. Bhattacherjee, B. Bhattacharjee, and M. Canim, "L-store: A real-time OLTP and OLAP system," in *Proceedings of the 21th International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018*, 2018, pp. 540–551. DOI: 10.5441/002/edbt.2018.65. [Online]. Available: https://doi.org/10.5441/002/edbt.2018.65.

[31] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood, "Implementation Techniques for Main Memory Database Systems," in *Proc. SIGMOD*, ACM, 1984, pp. 1–8, ISBN: 0-89791-128-8. DOI: 10.1145/602259.602261. [Online]. Available: http://dx.doi.org/10.1145/602259.602261.

[32] G. Weikum and G. Vossen, *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001.

[33] 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, Nov. 1976. DOI: 10.1145/360363.360369. [Online]. Available: http://doi.acm.org.ezproxy.lib.purdue.edu/10.1145/360363.360369.

[34] P. A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems," *ACM Comput. Surv.*, vol. 13, no. 2, pp. 185–221, Jun. 1981, ISSN: 0360-0300. DOI: 10.1145/356842.356846. [Online]. Available: http://dx.doi.org/10.1145/356842.356846.

[35] X. Yu, A. Pavlo, D. Sanchez, and S. Devadas, "TicToc: Time Traveling Optimistic Concurrency Control," in *Proc. SIGMOD*, ACM, 2016, pp. 1629–1642, ISBN: 978-1-4503-3531-7. DOI: 10.1145/2882903.2882935. [Online]. Available: http://dx.doi.org/10.1145/2882903.2882935.

[36] H. Kimura, "FOEDUS: OLTP Engine for a Thousand Cores and NVRAM," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15, Melbourne, Victoria, Australia: ACM, 2015, pp. 691–706, ISBN: 978-1-4503-2758-9. DOI: 10.1145/2723372.2746480. [Online]. Available: http://dx.doi.org/10.1145/2723372.2746480.

[37] K. Kim, T. Wang, R. Johnson, and I. Pandis, "ERMIA: Fast Memory-Optimized Database System for Heterogeneous Workloads," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16, San Francisco, California, USA: ACM, 2016, pp. 1675–1687, ISBN: 978-1-4503-3531-7. DOI: 10.1145/2882903.2882905. [Online]. Available: http://dx.doi.org/10.1145/2882903.2882905.

[38] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *Proc. SoCC*, ACM, 2010, pp. 143–154, ISBN: 978-1-4503-0036-0. DOI: 10.1145/1807128.1807152. [Online]. Available: http://dx.doi.org/10.1145/1807128.1807152.

[39] TPC, *TPC-C, On-Line Transaction Processing Benchmark, Version 5.11.0*. TPC Corporation, Feb. 2010. [Online]. Available: http://www.tpc.org/tpcc/.

[40] C. Curino, E. Jones, Y. Zhang, and S. Madden, "Schism: A Workload-driven Approach to Database Replication and Partitioning," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 48–57, Sep. 2010, ISSN: 2150-8097. DOI: 10.14778/1920841.1920853. [Online]. Available: http://dx.doi.org/10.14778/1920841.1920853.

[41] A. J. Bernstein, D. S. Gerstl, and P. M. Lewis, "Concurrency Control for Step-decomposed Transactions," *Inf. Syst.*, vol. 24, no. 9, pp. 673–698, Dec. 1999, ISSN: 0306-4379. [Online]. Available: http://portal.acm.org/citation.cfm?id=337922.

[42] V. Kumar, Ed., *Performance of Concurrency Control Mechanisms in Centralized Database Systems*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1995.

[43] A. Thomasian, "Concurrency Control: Methods, Performance, and Analysis," *ACM Comput. Surv.*, vol. 30, no. 1, pp. 70–119, Mar. 1998. DOI: 10.1145/274440.274443. [Online]. Available: http://doi.acm.org/10.1145/274440.274443.

[44] J. Giceva and M. Sadoghi, "Hybrid OLTP and OLAP," in *Encyclopedia of Big Data Technologies*, S. Sakr and A. Zomaya, Eds., Cham: Springer International Publishing, 2018, pp. 1–8, ISBN: 978-3-319-63962-8. DOI: . [Online]. Available: .

[45] T. Wang and H. Kimura, "Mostly-optimistic Concurrency Control for Highly Contended Dynamic Workloads on a Thousand Cores," *Proc. VLDB Endow.*, vol. 10, no. 2, pp. 49–60, Oct. 2016, ISSN: 2150-8097. DOI: 10.14778/3015274.3015276. [Online]. Available: http://dx.doi.org/10.14778/3015274.3015276.

[46] Y. Yuan, K. Wang, R. Lee, X. Ding, J. Xing, S. Blanas, and X. Zhang, "BCC: Reducing False Aborts in Optimistic Concurrency Control with Low Cost for In-memory Databases," *Proc. VLDB Endow.*, vol. 9, no. 6, pp. 504–515, Jan. 2016, ISSN: 2150-8097. DOI: 10.14778/2904121.2904126. [Online]. Available: http://dx.doi.org/10.14778/2904121.2904126.

[47] M. Sadoghi, M. Canim, B. Bhattacharjee, F. Nagel, and K. A. Ross, "Reducing Database Locking Contention Through Multi-version Concurrency," *Proc. VLDB Endow.*, vol. 7, no. 13, pp. 1331–1342, Aug. 2014, ISSN: 2150-8097. DOI: 10.14778/2733004.2733006. [Online]. Available: http://dx.doi.org/10.14778/2733004.2733006.

[48] H. T. Kung and J. T. Robinson, "On Optimistic Methods for Concurrency Control," *ACM Trans. Database Syst.*, vol. 6, no. 2, pp. 213–226, Jun. 1981, ISSN: 0362-5915. DOI: 10.1145/319566.319567. [Online]. Available: http://dx.doi.org/10.1145/319566.319567.

[49] P. Cincilla, S. Monnet, and M. Shapiro, "Gargamel: Boosting DBMS Performance by Parallelising Write Transactions," in *2012 IEEE 18th International Conference on Parallel and Distributed Systems*, Dec. 2012, pp. 572–579. DOI: 10.1109/ICPADS.2012.83.

[50] T. Qadah, S. Gupta, and M. Sadoghi, "Q-store: Distributed, multi-partition transactions via queue-oriented execution and communication," in *Proceedings of the 23rd International Conference on Extending Database Technology, EDBT 2020, Copenhagen, Denmark, March 30 - April 02, 2020*, A. Bonifati, Y. Zhou, M. A. V. Salles, A. Böhm, D. Olteanu, G. H. L. Fletcher, A. Khan, and B. Yang, Eds., OpenProceedings.org, 2020, pp. 73–84. DOI: 10.5441/002/edbt.2020.08. [Online]. Available: https://doi.org/10.5441/002/edbt.2020.08.

[51] J. N. Gray, "Notes on data base operating systems," in *Operating Systems: An Advanced Course*, ser. Lecture Notes in Computer Science, R. Bayer, R. M. Graham, and G. Seegmüller, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1978, pp. 393–481, ISBN: 978-3-540-35880-0. [Online]. Available: .

[52] E. P. 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, New York, NY, USA: ACM, 2010, pp. 603–614, ISBN: 978-1-4503-0032-2. DOI: 10.1145/1807167.1807233. [Online]. Available: http://doi.acm.org/10.1145/1807167.1807233.

[53] X. Yu, Y. Xia, A. Pavlo, D. Sánchez, L. Rudolph, and S. Devadas, "Sundial: Harmonizing Concurrency Control and Caching in a Distributed OLTP Database Management System," *PVLDB*, vol. 11, no. 10, pp. 1289–1302, 2018. DOI: 10.14778/3231751.3231763. [Online]. Available: http://www.vldb.org/pvldb/vol11/p1289-yu.pdf.

[54] 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*, ser. SIGMOD '16, New York, NY, USA: ACM, 2016, pp. 1659–1674, ISBN: 978-1-4503-3531-7. DOI: 10.1145/2882903.2882923. [Online]. Available: http://doi.acm.org/10.1145/2882903.2882923.

[55] Y. Lu, X. Yu, and S. Madden, "STAR: Scaling Transactions through Asymmetric Replication," *PVLDB*, vol. 12, no. 11, pp. 1316–1329, 2019. [Online]. Available: http://www.vldb.org/pvldb/vol12/p1316-lu.pdf.

[56] FaunaDB. (2019). "FaunaDB Website," [Online]. Available: https://fauna.com/.

[57] IBM. (Oct. 24, 2014). "DB2 Isolation levels," Knowlege Center, [Online]. Available: http://disq.us/t/2s92c84.

[58] Microsoft. (2019). "SQL Server Isolation Levels," [Online]. Available: https://docs.microsoft.com/en-us/sql/t-sql/statements/set-transaction-isolation-level-transact-sql.

[59] Oracle. (2019). "Data Concurrency and Consistency - 11g Release 2 (11.2)," [Online]. Available: .

[60] M. Sadoghi and S. Blanas, "Transaction Processing on Modern Hardware," *Synthesis Lectures on Data Management*, vol. 14, no. 2, pp. 1–138, Mar. 8, 2019, ISSN: 2153-5418. DOI: 10.2200/S00896ED1V01Y201901DTM058. [Online]. Available: https://www.morganclaypool.com/doi/10.2200/S00896ED1V01Y201901DTM058.

[61] S.-H. Wu, T.-Y. Feng, M.-K. Liao, S.-K. Pi, and Y.-S. Lin, "T-Part: Partitioning of Transactions for Forward-Pushing in Deterministic Database Systems," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16, New York, NY, USA: ACM, 2016, pp. 1553–1565, ISBN: 978-1-4503-3531-7. DOI: 10.1145/2882903.2915227. [Online]. Available: http://doi.acm.org/10.1145/2882903.2915227.

[62] R. Elmasri and S. B. Navathe, *Fundamentals of Database Systems*, 7th ed. Pearson, 2015, ISBN: 978-0-13-397077-7.

[63] D. J. Abadi and J. M. Faleiro, "An Overview of Deterministic Database Systems," *Commun. ACM*, vol. 61, no. 9, pp. 78–88, Aug. 2018, ISSN: 0001-0782. DOI: 10.1145/3181853. [Online]. Available: http://doi.acm.org/10.1145/3181853.

[64] H. A. Mahmoud, V. Arora, F. Nawab, D. Agrawal, and A. El Abbadi, "MaaT: Effective and scalable coordination of distributed transactions in the cloud," *Proceedings of the VLDB Endowment*, vol. 7, no. 5, pp. 329–340, Jan. 2014, ISSN: 21508097. DOI: 10.14778/2732269.2732270. [Online]. Available: http://dx.doi.org/10.14778/2732269.2732270.

[65] J. E. April, "A Scalable Concurrent malloc(3) Implementation for FreeBSD," 2006.

[66] Jemalloc. (2018). "Jemalloc Website," [Online]. Available: http://jemalloc.net/.

[67] C. Mohan, B. Lindsay, and R. Obermarck, "Transaction Management in the R* Distributed Database Management System," *ACM Trans. Database Syst.*, vol. 11, no. 4, pp. 378–396, Dec. 1986, ISSN: 0362-5915. DOI: 10.1145/7239.7266. [Online]. Available: http://doi.acm.org/10.1145/7239.7266.

[68] G. Samaras, K. Britton, A. Citron, and C. Mohan, "Two-Phase Commit Optimizations and Tradeoffs in the Commercial Environment," in *Proceedings of the Ninth International Conference on Data Engineering*, Washington, DC, USA: IEEE Computer Society, 1993, pp. 520–529, ISBN: 978-0-8186-3570-0. [Online]. Available: http://dl.acm.org/citation.cfm?id=645478.654794.

[69] B. W. Lampson and D. B. Lomet, "A New Presumed Commit Optimization for Two Phase Commit," in *Proceedings of the 19th International Conference on Very Large Data Bases*, ser. VLDB '93, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993, pp. 630–640, ISBN: 978-1-55860-152-9. [Online]. Available: http://dl.acm.org/citation.cfm?id=645919.672675.

[70] S. Das, D. Agrawal, and A. E. 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: ACM, 2010, pp. 163–174, ISBN: 978-1-4503-0036-0. DOI: 10.1145/1807128.1807157. [Online]. Available: http://dx.doi.org/10.1145/1807128.1807157.

[71] VoltDB. (2019). "VoltDB," [Online]. Available: https://www.voltdb.com/.

[72] M. T. Özsu and P. Valduriez, *Principles of Distributed Database Systems*, 4th ed. Springer International Publishing, 2020, ISBN: 978-3-030-26252-5. DOI: 10.1007/978-3-030-26253-2. [Online]. Available: https://www.springer.com/gp/book/9783030262525.

[73] B. Kemme, R. Jiménez-Peris, and M. Patiño-Martínez, *Database Replication*, ser. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2010. DOI: 10.2200/S00296ED1V01Y201008DTM007. [Online]. Available: https://doi.org/10.2200/S00296ED1V01Y201008DTM007.

[74] B. Kemme, R. Jiménez-Peris, M. Patiño-Martínez, and G. Alonso, "Database replication: A tutorial," in *Replication: Theory and Practice*, B. Charron-Bost, F. Pedone, and A. Schiper, Eds., ser. Lecture Notes in Computer Science, vol. 5959, Springer, 2010, pp. 219–252. DOI: .

[75] 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, Jun. 11, 2020, pp. 511–526, ISBN: 978-1-4503-6735-6. DOI: 10.1145/3318464.3389724. [Online]. Available: http://doi.org/10.1145/3318464.3389724.

[76] T. M. Qadah, "A queue-oriented transaction processing paradigm," in *Proceedings of the 20th International Middleware Conference Doctoral Symposium, Middleware 2019, Davis, CA, USA, December 09-13, 2019*, F. Nawab and E. Riviere, Eds., ACM, 2019, pp. 26–30. DOI: 10.1145/3366624.3368163. [Online]. Available: https://doi.org/10.1145/3366624.3368163.

[77] (). "Apache ZooKeeper," [Online]. Available: https://zookeeper.apache.org/.

[78] (). "Etcd," etcd, [Online]. Available: https://etcd.io/.

[79] (). "Apache Kafka," Apache Kafka, [Online]. Available: https://kafka.apache.org/.

[80] L. Lamport, "Paxos made simple, fast, and byzantine," in *Procedings of the 6th International Conference on Principles of Distributed Systems. OPODIS 2002, Reims, France, December 11-13, 2002*, A. Bui and H. Fouchal, Eds., ser. Studia Informatica Universalis, vol. 3, Suger, Saint-Denis, rue Catulienne, France, 2002, pp. 7–9.

[81] B. M. Oki and B. H. Liskov, "Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems," in *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '88, New York, NY, USA: Association for Computing Machinery, Jan. 1, 1988, pp. 8–17, ISBN: 978-0-89791-277-8. DOI: 10.1145/62546.62549. [Online]. Available: http://doi.org/10.1145/62546.62549.

[82] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIX ATC'14, USA: USENIX Association, Jun. 19, 2014, pp. 305–320, ISBN: 978-1-931971-10-2.

[83] *Google/snappy*, Google, May 8, 2021. [Online]. Available: https://github.com/google/snappy.

[84] F. P. Junqueira, B. C. Reed, and M. Serafini, "Zab: High-performance broadcast for primary-backup systems," in *2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN)*, Jun. 2011, pp. 245–256. DOI: 10.1109/DSN.2011.5958223.

[85] J. Gray, P. Helland, P. O'Neil, and D. Shasha, "The Dangers of Replication and a Solution," *SIGMOD Rec.*, vol. 25, no. 2, pp. 173–182, Jun. 1996, ISSN: 0163-5808. DOI: 10.1145/235968.233330. [Online]. Available: http://dx.doi.org/10.1145/235968.233330.

[86] B. Kemme and G. Alonso, "A new approach to developing and implementing eager database replication protocols," *ACM Transactions on Database Systems*, vol. 25, no. 3, pp. 333–379, Sep. 1, 2000, ISSN: 0362-5915. DOI: 10.1145/363951.363955. [Online]. Available: http://doi.org/10.1145/363951.363955.

[87] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso, "Understanding replication in databases and distributed systems," in *Proceedings 20th IEEE International Conference on Distributed Computing Systems*, Apr. 2000, pp. 464–474. DOI: 10.1109/ICDCS.2000.840959.

[88] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems*, vol. 16, no. 2, pp. 133–169, May 1, 1998, ISSN: 0734-2071. DOI: 10.1145/279227.279229. [Online]. Available: http://doi.org/10.1145/279227.279229.

[89] A. Thomson and D. J. Abadi, "The Case for Determinism in Database Systems," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 70–80, Sep. 2010, ISSN: 2150-8097. DOI: 10.14778/1920841.1920855. [Online]. Available: http://dx.doi.org/10.14778/1920841.1920855.

[90] S. Gupta, J. Hellings, S. Rahnama, and M. Sadoghi, "Proof-of-execution: Reaching consensus through fault-tolerant speculation," in *Proceedings of the 24th International Conference on Extending Database Technology, EDBT 2021, Nicosia, Cyprus, March 23 - 26, 2021*, Y. Velegrakis, D. Zeinalipour-Yazti, P. K. Chrysanthis, and F. Guerra, Eds., OpenProceedings.org, 2021, pp. 301–312. DOI: 10.5441/002/edbt.2021.27. [Online]. Available: https://doi.org/10.5441/002/edbt.2021.27.

[91] F. Nawab and M. Sadoghi, "Blockplane: A global-scale byzantizing middleware," in *35th IEEE International Conference on Data Engineering*, 2019, pp. 124–135. DOI: 10.1109/ICDE.2019.00020.

[92] S. Gupta, J. Hellings, S. Rahnama, and M. Sadoghi, "Building high throughput permissioned blockchain fabrics: Challenges and opportunities," *Proc. VLDB Endow.*, vol. 13, no. 12, pp. 3441–3444, 2020. DOI: 10.14778/3415478.3415565. [Online]. Available: http://www.vldb.org/pvldb/vol13/p3441-gupta.pdf.

[93] S. Gupta, J. Hellings, and M. Sadoghi, "RCC: Resilient concurrent consensus for high-throughput secure transaction processing," in *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*, IEEE, 2021, pp. 1392–1403. DOI: 10.1109/ICDE51399.2021.00124. [Online]. Available: https://doi.org/10.1109/ICDE51399.2021.00124.

[94] S. Gupta, J. Hellings, and M. Sadoghi, *Fault-Tolerant Distributed Transactions on Blockchain*, ser. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2021. DOI: 10.2200/S01068ED1V01Y202012DTM065. [Online]. Available: https://doi.org/10.2200/S01068ED1V01Y202012DTM065.

[95] S. Rahnama, S. Gupta, T. Qadah, J. Hellings, and M. Sadoghi, "Scalable, resilient and configurable permissioned blockchain fabric," *Proc. VLDB Endow.*, vol. 13, no. 12, pp. 2893–2896, 2020. [Online]. Available: http://www.vldb.org/pvldb/vol13/p2893-rahnama.pdf.

[96] A. R. Mahmood, A. M. Aly, T. Qadah, E. K. Rezig, A. Daghistani, A. Madkour, A. S. Abdelhamid, M. S. Hassan, W. G. Aref, and S. Basalamah, "Tornado: A distributed spatio-textual stream processing system," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 2020–2023, Aug. 1, 2015, ISSN: 2150-8097. DOI: 10.14778/2824032.2824126. [Online]. Available: http://doi.org/10.14778/2824032.2824126.

[97] A. M. Aly, A. R. Mahmood, M. S. Hassan, W. G. Aref, M. Ouzzani, H. Elmeleegy, and T. Qadah, "AQWA: Adaptive query workload aware partitioning of big spatial data," *Proceedings of the VLDB Endowment*, vol. 8, no. 13, pp. 2062–2073, Sep. 1, 2015, ISSN: 2150-8097. DOI: 10.14778/2831360.2831361. [Online]. Available: http://doi.org/10.14778/2831360.2831361.

[98] A. S. Abdelhamid, M. Tang, A. M. Aly, A. R. Mahmood, T. Qadah, W. G. Aref, and S. M. Basalamah, "Cruncher: Distributed in-memory processing for location-based services," in *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*, IEEE Computer Society, 2016, pp. 1406–1409. DOI: 10.1109/ICDE.2016.7498356. [Online]. Available: https://doi.org/10.1109/ICDE.2016.7498356.

VITA

Thamir M. Qadah completed his Bachelor of Science in Computer Engineering from King Fahd University of Petroleum and Minerals in 2005 and a Master of Science in Computer Science from California State University at Fullerton in 2008. He earned his Master of Science in Electrical and Computer Engineering from the School of Electrical and Computer Engineering at Purdue University in 2014 and completed a postgraduate certification in Teaching and Learning in Engineering from the School of Engineering Education at Purdue University in 2020.

Thamir worked as a software engineer at Zykis LLC, a start-up specializing in providing information technology services for the automotive industry, from 2008 to 2010. After that he worked as a lecturer from 2010 to 2011 in the Computer Sciences department at Umm Al-Qura University. He has been a research associate with Exploratory Systems Lab since 2017. In 2019, he interned at Datometry, a start-up specializing in database management systems virtualization in cloud computing environments.

Since 2015, He has been active in the research community as a reviewer for top-tier conferences such as SIGMOD, VLDB, ICDE, ICDCS, ATC, EDBT, Middleware, and CIKM, as well as a reviewer for journals like IEEE Access. Moreover, he served as a committee member of the artifact evaluation committee for SOSP, OSDI, and ASPLOS.

His research interests include designing and implementing secure, dependable, and high-performance software systems that exploit modern hardware technologies. His research on queue-oriented transaction processing is recognized by the Best Paper Award in Middleware'18.