Q-Store: Distributed, Multi-partition Transactions via Queue-oriented Execution and Communication

Thamir M. Qadah[1†], Suyash Gupta[2], Mohammad Sadoghi[2]
Exploratory Systems Lab
[1]Purdue University, West Lafayette
[2]University of California, Davis
[†] Umm Al-Qura University, Makkah
tqadah@purdue.edu, sugupta@ucdavis.edu, msadoghi@ucdavis.edu

ABSTRACT

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.

In this paper, 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.

1 INTRODUCTION

Distributed transaction processing is challenging due to the inherent overheads of costly commit protocols like *2-Phase-Commit* (2PC) [13]. Even for use cases such as in-memory databases and stored-procedure-based transactions, 2PC is either used (e.g., [24, 25, 46]) or avoided by eliminating the processing of multi-partitioned transactions (e.g., [27, 28]). 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.

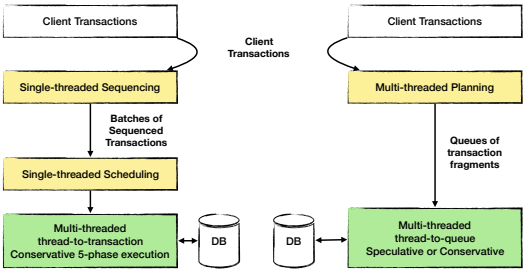


Figure 1: Overview of transaction processing in Calvin (left) and Q-Store (right)

Deterministic databases [12, 40] 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 [20]. Second, they mostly support a conservative (non-speculative) form of transaction execution. One exception is the work by Jones et al. [24], 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 [34, 35].

In this paper, 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., [22, 30, 33, 37, 38]).

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 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 paper.

- 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).
- 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 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 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 6).

2 BACKGROUND

In this section, we give an overview of Calvin [40] as a representative for deterministic databases. As far as we know, Calvin is regarded as the state-of-the-art distributed deterministic transaction processing protocol, and has been commercialized [12]. Other deterministic transaction processing protocols are either designed for non-distributed environments (e.g., [9, 11, 35]) or a variation that improves parts of Calvin's protocol while re-using the remaining parts as-is (e.g., [44]). These proposals are covered in Section 7 in more details. We also briefly describe the transaction model used by Q-Store which adopts the same transaction model used by [35], which is in sharp contrast from Calvin's transaction model.

2.1 Transaction Processing in Calvin

This section gives a brief description of how Calvin works based on [40]. The basic processing flow requires 3 phases: a sequencing phase, a scheduling phase, and an execution phase with 5 sub-phases. Figure 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* [8]. 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. [40] 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 [20], 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.

2.2 Q-Store's Transaction Model

We adopt the same transaction model used by [35]. In this model, a transaction is broken into fragments. A fragment can perform

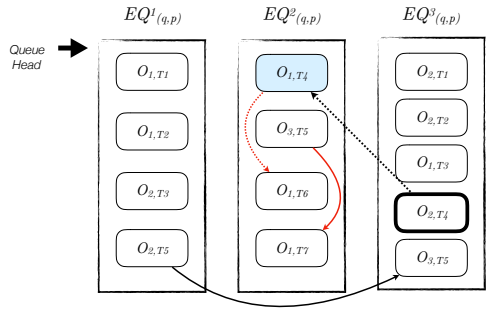


Figure 2: An example illustrating transaction dependencies in Q-Store. Execution-queues (EQs) are planned by planning-thread $PT_{(q,p)}$

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.

Furthermore, there are can be dependencies among fragments. In Figure 2, we illustrate these dependencies. There are 7 planned transactions in 3 execution-queues. Fragments are denoted as O_{i,T_j} where i denotes the fragment index in transaction T_j. We describe the notations in detail in Section 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 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 1, we show the critical differences between Calvin's processing model and Q-Store' processing model. On the left side (Calvin) of Figure 1, the total number of phases required to process a batch of

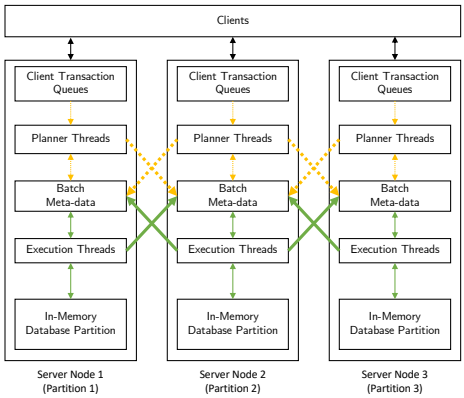


Figure 3: System Architecture

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

3.1 Queue-oriented Architecture

In Figure 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. Mechanisms for load-balancing include client-side libraries and middle-ware-based mechanisms. These details are beyond the scope of this paper.

Also in Figure 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, yellow arrows depict communication patterns during the planning-phase

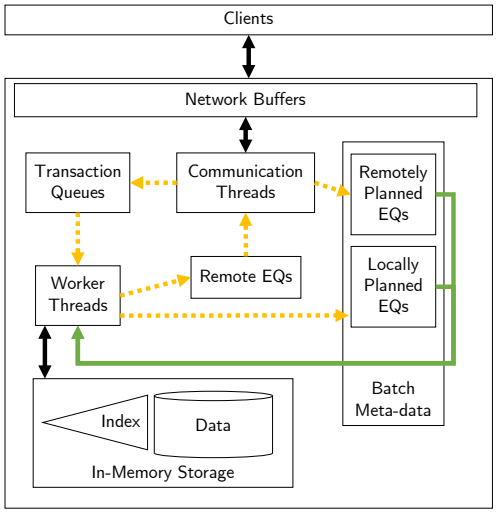


Figure 4: Server Node Architecture

while green arrows depict communication patterns during the execution-phase.

Zooming into a single node, Figure 4, we illustrate the major components of a server node. Similar to Figure 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.

3.2 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.

$$\mathbb{P}_q := \{P_{q,1},\ P_{q,2}, ..., P_{q,p}\},\ \text{where } q, p \geq 1 \quad (1)$$

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 Logging and Recovery

Q-Store like other deterministic transaction processing systems (e.g., [25, 40]) assumes a deterministic stored procedure based transaction model [1]. 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 paper, we mainly focus on transaction execution as we can rely on the same techniques for logging and recovery as [25, 40].

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.

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 paper.

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

$$\mathbb{Q}_{q,p} := \{Q^1_{q,p},\ Q^2_{q,p}, ..., Q^i_{q,p}\},\ \text{where } i \geq 1 \quad (2)$$

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 $Q^i_{q,p}$ 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:

$$\forall\ P_{q,p}, \in \mathbb{P}_q,\ |\mathbb{Q}_{q,p}| \geq |\mathbb{E}| \tag{3}$$

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

$$O_T := \{O_{1,T},\ O_{2,T}, ..., O_{k,T}\}$$

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.

$$\forall\ i, j,\ i < j \rightarrow T_{(j,p,q)} \xrightarrow{\text{follows}} T_{(i,p,q)} \tag{4}$$

Equation 4 shows that as $T_{(i,p,q)}$ has a smaller identifier (i) than $T_{(j,p,q)}$, so it must have been placed in the client-transaction-queue before $T_{(j,p,q)}$.

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 $Q^i_{q,p}$ 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.

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 $\mathbb{E}$.

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

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.*

$$\begin{aligned} &\forall\ Q_m \in \mathbb{Q}_{q,p},\ \forall\ Q_n \in \mathbb{Q}_{s,t},\ \forall\ O_i \in Q_m,\ \forall\ O_j \in Q_n \\ &|\ (q > s)\ \vee\ ((q = s)\ \wedge\ (p > t)) \rightarrow O_j \xrightarrow{\text{follows}} O_i \end{aligned} \tag{5}$$

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 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.

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 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.

4.4 Serializability

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

Theorem 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: $O_{i,T_n}, O_{j,T_n}, O_{k,T_m}$ 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,

$$\exists Q_a, Q_b \text{ s.t. } \{O_{i,T_n}, O_{k,T_m}\} \in Q_a \wedge \{O_{j,T_n}, O_{l,T_m}\} \in Q_b \tag{6}$$

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

$$O_{k,T_m} \xrightarrow{\text{follows}} O_{i,T_n} \wedge O_{j,T_n} \xrightarrow{\text{follows}} O_{l,T_m} \tag{7}$$

The other non-serializable history is captured by:

$$O_{i,T_n} \xrightarrow{\text{follows}} O_{k,T_m} \wedge O_{l,T_m} \xrightarrow{\text{follows}} O_{j,T_n} \tag{8}$$

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.

□

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 $Q'^{j}_{q,p}$ such that they only contain read-only transaction fragments as shown in Eq. 9. Read-only transaction fragments do not perform any write operations.

$$\begin{aligned} &\forall\, P_{q,p}, \in \mathbb{P}_q, \mathbb{Q}_{q,p} := \{Q^1_{q,p},\ Q^2_{q,p}, ..., Q^i_{q,p}\} \\ &\cup \{Q'^1_{q,p},\ Q'^2_{q,p}, ..., Q'^j_{q,p}\},\ \text{where } i \geq 1, j \geq 1 \end{aligned} \tag{9}$$

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.

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 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.

Limitations Advantages and disadvantages of deterministic transaction processing are discussed in the literature [36]. The key limitation of deterministic transaction processing is that the knowledge of the full read/write sets is required. One approach is to run the transaction without committing its write-set to compute the full read/write sets [40]. 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 study of these approach is beyond the scope of this paper.

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, 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 and 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 [7].

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.

6 EVALUATION

In this section, we present an extensive evaluation of Q-Store. We implement our techniques in ExpoDB [18, 19, 35]. 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 [3].
- TIMESTAMP: A basic time-ordering protocol [3] 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 [29] that is a representative of optimistic concurrency control protocols.
- Calvin: A deterministic transaction processing protocol [40].

We use a range-based partitioning instead of the original hash-based partitioning used by [20].

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.

Table 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), 0.4, 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%

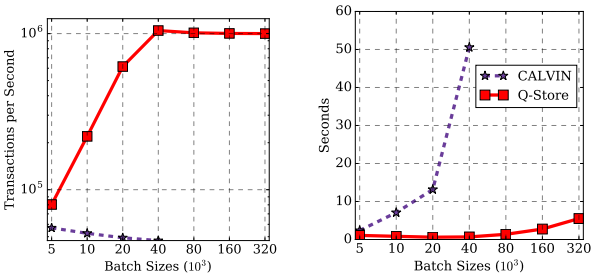
We use Ubuntu 16.04 (xenial), GCC 5.4, Jemalloc 4.5.0 [2, 23] and compile our code with -O2 compiler optimization flag. We pin threads to cores to reduce the variance from the operating system scheduling and the effect of the caching system. Each dedicates 4 threads as worker threads, and 4 as communication 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 [5]. 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 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 [41]. 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[20, 45], 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 [41].

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 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 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?



(a) Throughput **(b) 99^{th} Percentile Latency**

Figure 5: Impact of varying batch sizes on the system throughput and 99^{th} percentile latency of deterministic systems.

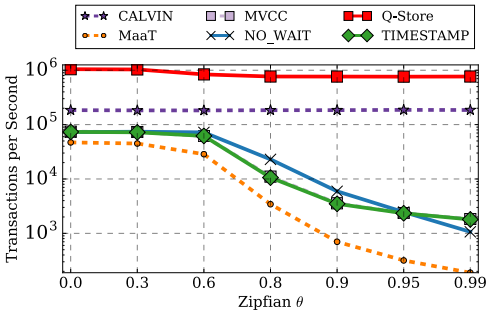


Figure 6: Impact of varying the data access skewness parameter θ of the Zipfian distribution on systems throughput (log scale).

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 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 $5ms$ [20]. 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 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.

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.

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 7 are for low contention (i.e., $\theta = 0.0$). Note that in comparison to Figure 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

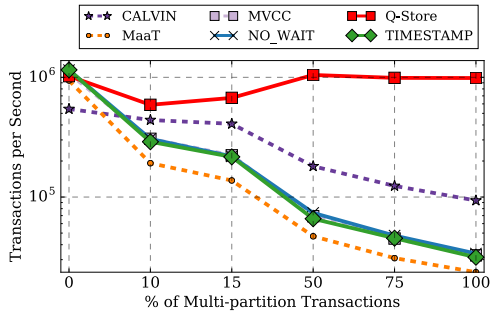


Figure 7: Impact of varying the percentage of multi-partitions transactions in the workload on the system's throughput.

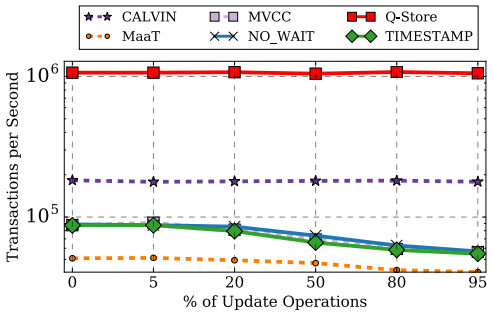


Figure 8: Impact of varying the percentage of update operations in the workload on the system's throughput.

thus, their throughput decreases. Thus, our results validate previously published results (e.g., [20]), 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. 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 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.

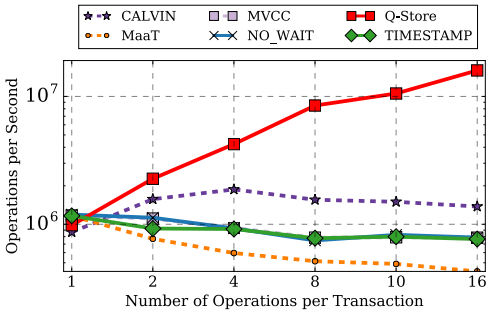


Figure 9: The impact of varying the number of operation per transaction on system's throughput. We force each operation access a different partition. This results is for low contention $\theta = 0.0$.

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 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.

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 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.

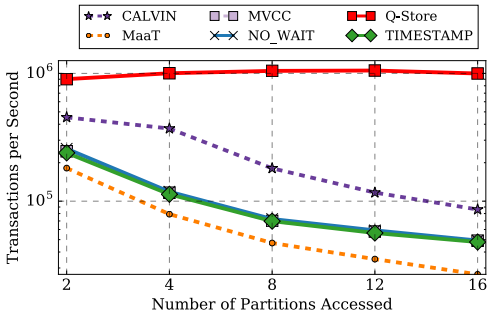


Figure 10: The impact of varying the number of partitions accessed by each transaction on the system's throughput.

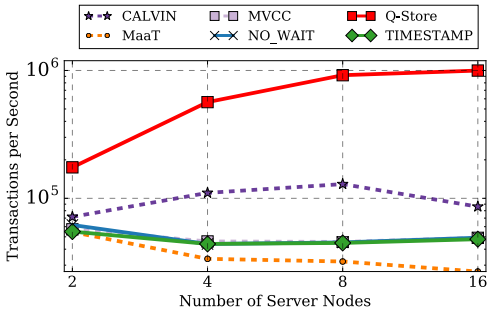


Figure 11: Throughput scalability results while varying the number of server nodes.

Vary partitions per transaction In Figure 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 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 11, shows that Q-Store scales well as the number of server nodes increases in the cluster, achieving over

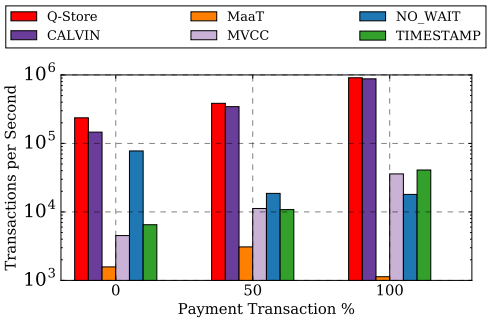


Figure 12: The impact of different TPC-C transaction mixes on the system's throughput. 15% **multi-partition transactions is used.**

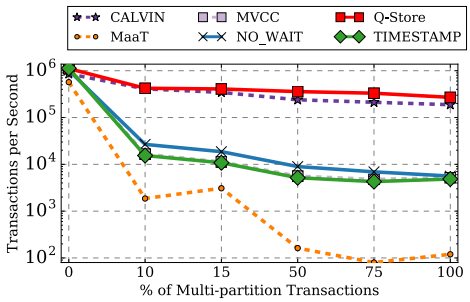


Figure 13: Varying the percentage of multi-partition transaction with equal ratios of Payment and NewOrder transactions.

1 million transactions per second at 16 server nodes. 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.

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 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 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 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]

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 Goodman [3] 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., [26, 29, 31, 39]) 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 [1]. 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 [6] 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 [27] 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., [12, 42]. For single-partitioned workloads, H-Store[25] 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. [24] 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 [4], 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 [40, 44] uses determinism to eliminate the cost of two-phase-commit protocol when processing distributed transactions. T-Part [44] 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., [10, 34, 35, 43]). Q-Store goes beyond these proposals and exploits intra-transaction parallelism within and across nodes in the context of distributed transaction processing.

8 CONCLUSIONS AND FUTURE WORK

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.

There are renewed research interests in byzantine fault-tolerance for transaction processing [14–17, 21, 32]. In future, we plan to support byzantine fault-tolerance for database transactions in Q-Store. On the one hand, blockchain transactions are deterministic, which aligns with the kind of transactions that Q-Store's supports. On the other hand, it is very challenging to design and implement

[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 paper, and we leave it to future work.

efficient, Byzantine fault-tolerant protocols. We believe that the design principles behind Q-Store can lead to efficient Byzantine fault-tolerant protocols, as very few blockchain proposals look at optimizing execution.

REFERENCES

[1] Daniel J. Abadi and Jose M. Faleiro. 2018. An Overview of Deterministic Database Systems. *Commun. ACM* 61, 9 (Aug. 2018), 78–88. https://doi.org/10.1145/3181853

[2] Jason Evans April. 2006. A Scalable Concurrent Malloc(3) Implementation for FreeBSD.

[3] Philip A. Bernstein and Nathan Goodman. 1981. Concurrency Control in Distributed Database Systems. *ACM Comput. Surv.* 13, 2 (June 1981), 185–221. https://doi.org/10.1145/356842.356846

[4] P. Cincilla, S. Monnet, and M. Shapiro. 2012. Gargamel: Boosting DBMS Performance by Parallelising Write Transactions. In *Proc. ICPADS'12*. 572–579. https://doi.org/10.1109/ICPADS.2012.83

[5] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proc. SoCC'10*. ACM, 143–154. https://doi.org/10.1145/1807128.1807152

[6] Sudipto Das, Divyakant Agrawal, and Amr E. Abbadi. 2010. G-Store: A Scalable Data Store for Transactional Multi Key Access in the Cloud. In *Proc. SoCC'10 (SoCC '10)*. ACM, Indianapolis, Indiana, USA, 163–174. https://doi.org/10.1145/1807128.1807157

[7] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael R. Stonebraker, and David A. Wood. 1984. Implementation Techniques for Main Memory Database Systems. In *Proc. SIGMOD'84*. ACM, 1–8. https://doi.org/10.1145/602259.602261

[8] Ramez Elmasri and Shamkant B. Navathe. 2015. *Fundamentals of Database Systems* (7th ed.). Pearson.

[9] Jose M. Faleiro and Daniel J. Abadi. 2015. Rethinking Serializable Multiversion Concurrency Control. *Proc. VLDB Endow.* 8, 11 (July 2015), 1190–1201. https://doi.org/10.14778/2809974.2809981

[10] Jose M. Faleiro, Daniel J. Abadi, and Joseph M. Hellerstein. 2017. High Performance Transactions via Early Write Visibility. *Proc. VLDB Endow.* 10, 5 (Jan. 2017), 613–624. https://doi.org/10.14778/3055540.3055553

[11] Jose M. Faleiro, Alexander Thomson, and Daniel J. Abadi. 2014. Lazy Evaluation of Transactions in Database Systems. In *Proc. SIGMOD'14*. ACM, 15–26. https://doi.org/10.1145/2588555.2610529

[12] FaunaDB. 2019. FaunaDB Website. https://fauna.com/. (2019).

[13] J. N. Gray. 1978. Notes on Data Base Operating Systems. In *Operating Systems: An Advanced Course*, R. Bayer, R. M. Graham, and G. Seegmüller (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 393–481.

[14] Suyash Gupta, Jelle Hellings, Sajjad Rahnama, and Mohammad Sadoghi. 2019. Proof-of-Execution: Reaching Consensus through Fault-Tolerant Speculation. *CoRR* abs/1911.00838 (2019).

[15] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. 2019. Brief Announcement: Revisiting Consensus Protocols through Wait-Free Parallelization. In *DISC'19*. 44:1–44:3. https://doi.org/10.4230/LIPIcs.DISC.2019.44

[16] Suyash Gupta, Sajjad Rahnama, and Mohammad Sadoghi. 2019. Permissioned Blockchain Through the Looking Glass: Architectural and Implementation Lessons Learned. *CoRR* abs/1911.09208 (2019).

[17] Suyash Gupta and Mohammad Sadoghi. 2018. Blockchain Transaction Processing. In *Encyclopedia of Big Data Technologies*. Springer International Publishing, Cham, 1–11. https://doi.org/10.1007/978-3-319-63962-8_333-1

[18] Suyash Gupta and Mohammad Sadoghi. 2018. EasyCommit: A Non-Blocking Two-Phase Commit Protocol. In *Proc. EDBT'18*. https://doi.org/10.5441/002/edbt.2018.15

[19] Suyash Gupta and Mohammad Sadoghi. 2019. Efficient and non-blocking agreement protocols. *Distributed and Parallel Databases* (13 Apr 2019). https://doi.org/10.1007/s10619-019-07267-w

[20] Rachael Harding, Dana Van Aken, Andrew Pavlo, and Michael Stonebraker. 2017. An Evaluation of Distributed Concurrency Control. *Proc. VLDB Endow.* 10, 5 (Jan. 2017), 553–564. https://doi.org/10.14778/3055540.3055548

[21] Jelle Hellings and Mohammad Sadoghi. 2019. Brief Announcement: The Fault-Tolerant Cluster-Sending Problem. In *DISC'19*. 45:1–45:3. https://doi.org/10.4230/LIPIcs.DISC.2019.45

[22] IBM. 2014. DB2 Isolation Levels. http://disq.us/t/2s92c84. (Oct. 2014).

[23] Jemalloc. 2018. Jemalloc Website. http://jemalloc.net/. (2018).

[24] Evan P.C. Jones, Daniel J. Abadi, and Samuel Madden. 2010. Low Overhead Concurrency Control for Partitioned Main Memory Databases. In *Proc. SIGMOD'10*. ACM, New York, NY, USA, 603–614. https://doi.org/10.1145/1807167.1807233

[25] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. 2008. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. *Proc. VLDB Endow.* 1, 2 (Aug. 2008), 1496–1499. https://doi.org/10.14778/1454159.1454211

[26] Butler W. Lampson and David B. Lomet. 1993. A New Presumed Commit Optimization for Two Phase Commit. In *Proc. VLDB'93*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 630–640.

[27] Qian Lin, Pengfei Chang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Zhengkui Wang. 2016. Towards a Non-2PC Transaction Management in Distributed Database Systems. In *Proc. SIGMOD'16*. ACM, New York, NY, USA, 1659–1674. https://doi.org/10.1145/2882903.2882923

[28] Yi Lu, Xiangyao Yu, and Samuel Madden. 2019. STAR: Scaling Transactions through Asymmetric Replication. *PVLDB* 12, 11 (2019), 1316–1329.

[29] Hatem A. Mahmoud, Vaibhav Arora, Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. 2014. MaaT: Effective and Scalable Coordination of Distributed Transactions in the Cloud. *PVLDB* 7, 5 (Jan. 2014), 329–340. https://doi.org/10.14778/2732269.2732270

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

[31] C. Mohan, B. Lindsay, and R. Obermarck. 1986. Transaction Management in the R* Distributed Database Management System. *ACM Trans. Database Syst.* 11, 4 (Dec. 1986), 378–396. https://doi.org/10.1145/7239.7266

[32] Faisal Nawab and Mohammad Sadoghi. 2019. Blockplane: A Global-Scale Byzantizing Middleware. In *ICDE'19*. 124–135. https://doi.org/10.1109/ICDE.2019.00020

[33] Oracle. 2019. Data Concurrency and Consistency - 11g Release 2 (11.2). https://docs.oracle.com/cd/E25054_01/server.1111/e25789/consist.htm. (2019).

[34] Ippokratis Pandis, Ryan Johnson, Nikos Hardavellas, and Anastasia Ailamaki. 2010. Data-Oriented Transaction Execution. *Proc. VLDB Endow.* 3, 1-2 (Sept. 2010), 928–939. https://doi.org/10.14778/1920841.1920959

[35] Thamir M. Qadah and Mohammad Sadoghi. 2018. QueCC: A Queue-Oriented, Control-Free Concurrency Architecture. In *Proc. Middleware '18*. ACM, Rennes, France, 13–25. https://doi.org/10.1145/3274808.3274810

[36] Kun Ren, Alexander Thomson, and Daniel J. Abadi. 2014. An Evaluation of the Advantages and Disadvantages of Deterministic Database Systems. *Proc. VLDB Endow.* 7, 10 (June 2014), 821–832. https://doi.org/10.14778/2732951.2732955

[37] Mohammad Sadoghi, Souvik Bhattacherjee, Bishwaranjan Bhattacharjee, and Mustafa Canim. 2018. L-Store: A Real-Time OLTP and OLAP System. In *Proc. EDBT'18*. 540–551. https://doi.org/10.5441/002/edbt.2018.65

[38] Mohammad Sadoghi and Spyros Blanas. 2019. Transaction Processing on Modern Hardware. *Synthesis Lectures on Data Management* 14, 2 (March 2019), 1–138. https://doi.org/10.2200/S00896ED1V01Y201901DTM058

[39] George Samaras, Kathryn Britton, Andrew Citron, and C. Mohan. 1993. Two-Phase Commit Optimizations and Tradeoffs in the Commercial Environment. In *Proc. ICDE'93*. IEEE Computer Society, Washington, DC, USA, 520–529.

[40] Alexander Thomson, Thaddeus Diamond, Shu C. Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proc. SIGMOD'12*. ACM, 1–12. https://doi.org/10.1145/2213836.2213838

[41] TPC. 2010. *TPC-C, On-Line Transaction Processing Benchmark, Version 5.11.0*. TPC Corporation.

[42] VoltDB. 2019. VoltDB. https://www.voltdb.com/. (2019).

[43] Zhaoguo Wang, Shuai Mu, Yang Cui, Han Yi, Haibo Chen, and Jinyang Li. 2016. Scaling Multicore Databases via Constrained Parallel Execution. In *Proc. SIGMOD'16 (SIGMOD '16)*. ACM, New York, NY, USA, 1643–1658. https://doi.org/10.1145/2882903.2882934

[44] Shan-Hung Wu, Tsai-Yu Feng, Meng-Kai Liao, Shao-Kan Pi, and Yu-Shan Lin. 2016. T-Part: Partitioning of Transactions for Forward-Pushing in Deterministic Database Systems. In *Proc. SIGMOD'16*. ACM, New York, NY, USA, 1553–1565. https://doi.org/10.1145/2882903.2915227

[45] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *Proc. VLDB Endow.* 8, 3 (Nov. 2014), 209–220. https://doi.org/10.14778/2735508.2735511

[46] Xiangyao Yu, Yu Xia, Andrew Pavlo, Daniel Sánchez, Larry Rudolph, and Srinivas Devadas. 2018. Sundial: Harmonizing Concurrency Control and Caching in a Distributed OLTP Database Management System. *PVLDB* 11, 10 (2018), 1289–1302. https://doi.org/10.14778/3231751.3231763