QueCC: A Queue-oriented, Control-free Concurrency Architecture

Thamir M. Qadah[1], Mohammad Sadoghi[2]
Exploratory Systems Lab
[1]Purdue University, West Lafayette
[2]University of California, Davis
tqadah@purdue.edu,msadoghi@ucdavis.edu

ABSTRACT

We investigate a coordination-free approach to transaction processing on 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. 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 out-perform 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.

CCS CONCEPTS

• **Information systems → Data management systems**; **DBMS engine architectures**; **Database transaction processing**; **Parallel and distributed DBMSs**; **Key-value stores**; **Main memory engines**;

ACM Reference Format:
Thamir M. Qadah[1], Mohammad Sadoghi[2]. 2018. QueCC: A Queue-oriented, Control-free Concurrency Architecture. In *19th International Middleware Conference (Middleware '18), December 10–14, 2018, Rennes, France.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3274808.3274810

1 INTRODUCTION

New multi-socket, many-core hardware architectures with tens or hundreds of cores are becoming commonplace in the market today [16, 27, 33]. This is a trend that is expected to increase exponentially, thus, reaching thousands of cores per box in the near future [17]. 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 [26, 28, 39, 44, 45]. 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 [9, 11, 15, 19, 36, 37]. 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 [29, 44] 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. [29] 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. [44] 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 paper is motivated by a simple profound question: *is it possible to have concurrent execution over shared*

Middleware '18, December 10–14, 2018, Rennes, France

ACM ISBN 978-1-4503-5702-9/18/12…$15.00
https://doi.org/10.1145/3274808.3274810

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.

1.1 Emergence of Deterministic Data Stores

Early proposals for deterministic execution for transaction processing aimed at data replication (e.g., [18, 20]). 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. [19]) 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 [37]. 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 [30, 43] 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 [30]. 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 [9, 11, 15, 19, 36, 37].

1.2 Contributions

In this paper, 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).
- 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 3).
- we design a novel two-phase, priority-based, queue-oriented planning and execution model that eliminates the need for concurrency control (Section 4).
- we prototype our proposed concurrency architecture within ExpoDB [13, 14], 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 5).

2 FORMALISM

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

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[1].

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 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 [10]. 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 $v_j = v_i + 1$. 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. ***(Intra-transaction data dependency):***
An intra-transaction data dependency exist between two transaction fragments f_i and f_j, denoted as $f_i \overset{d}{\rightarrow} f_j$, 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 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 4. ***(Intra-transaction commit dependency):***
An intra-transaction commit dependency exist between two transaction fragments f_i and f_j, denoted as $f_i \overset{c}{\rightarrow} f_j$, 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 [10, 34, 41], the notion of actions in DORA[28], and the notion of record actions in LADS [44]. 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 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 $E_{t_i} = \{(f_p, f_q) | f_p \overset{d}{\rightarrow} f_q \vee f_p \overset{c}{\rightarrow} f_q\}$

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 [25] when not considering the notion of transaction fragments. They are the result of speculative reading of uncommitted records [31]. We formally define them as follows:

Definition 6. ***(Inter-transaction commit dependency):***
An inter-transaction commit dependency exist between two transaction fragments f_i and f_j is denoted as $f_i \overset{s}{\rightarrow} f_j$, 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.[10].

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.

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 1) is geared towards a throughput-optimized in-memory stores.

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

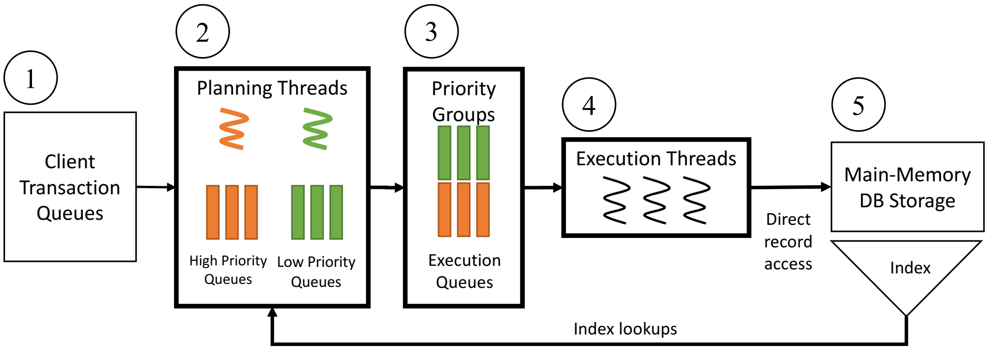


Figure 1: Overview of Priority-based, Queue-oriented Architecture

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 7. ***(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 [7].

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 [42]. 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 1).

Theorem 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. □

4 CONTROL-FREE ARCHITECTURAL DESIGN

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

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 predetermined 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 8. ***(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 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.

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, which has the following logic: $f_i = \{a = read(k_i)\}, f_j = \{b = a+1; write(k_j, b)\}$, where keys are denoted as k_i. In this transaction,

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

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

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 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 [25, 31] 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 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.

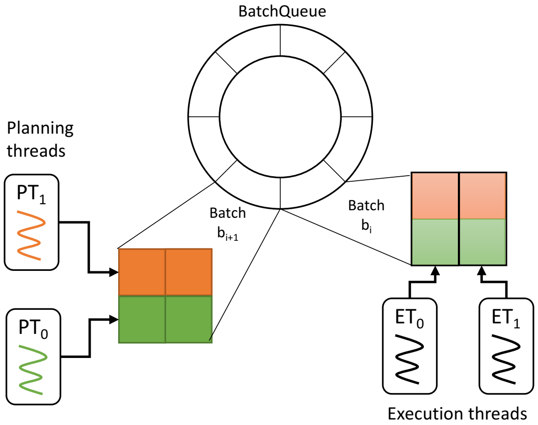


Figure 2: Example of concurrent batch planning and execution with 4 worker threads (2 planner threads + 2 execution threads). Priority groups are color-coded by planners. Execution threads process transactions from both priority groups.

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, 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

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

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.

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[10]).

Table 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%

However, QueCC is not limited to only SWV and can support multiple write visibility policies. Faleiro et al. [10] 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 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.

5 EXPERIMENTAL ANALYSIS

We have evaluated the QueCC protocol in our ExpoDB platform [13, 14]. 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 [13, 14]. ExpoDB's comprehensive concurrency testbed includes a variant of two-phase locking [8] (i.e., NO-WAIT [3] as a representative of pessimistic concurrency control), TicToc [46], Cicada [26], SILO [39], FOEDUS with MOCC [22, 40], ERMIA with SSI and SSN [21], and H-Store [19], all of which were compared against QueCC.

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

Table 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%

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).

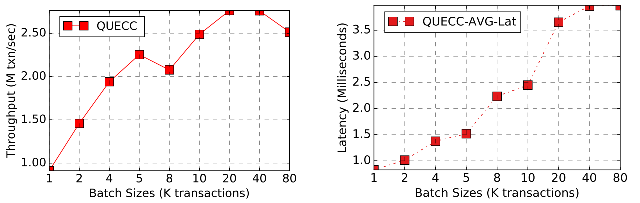
5.2 Workloads Overview

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

YCSB[5] 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 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, 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 [38] 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 [6]. Similar to previous studies in the literature[15, 45], 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 [38]. 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 paper, 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 [22, 25, 26, 39, 46] 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 3: Varying batch sizes and high data access skew ($\theta = 0.99$)

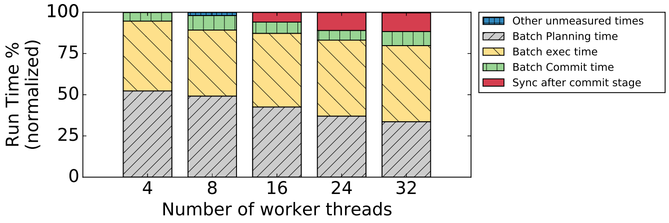


Figure 4: Time breakdown when varying number of worker threads.

5.3 YCSB Experiments

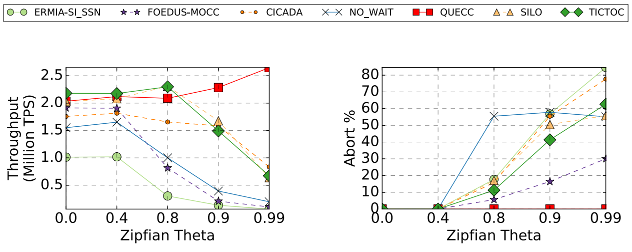
Using YCSB workloads, we start by evaluating the performance of QueCC with different batch sizes, which is a unique aspect of QueCC. Subsequently, we compare QueCC with other concurrency control protocols.

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 3b, which is considered reasonable for many applications. For the remaining experiments, we use a batch of size $10K$.

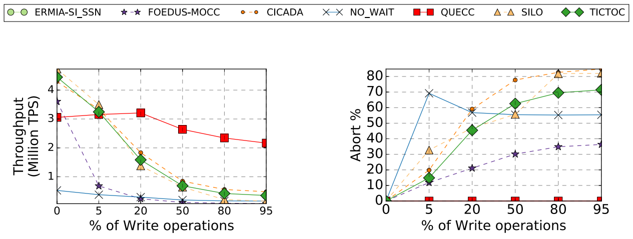
Time Breakdown Figure 4 illustrates the time breakdown spent on each phase of QueCC under highly skewed data accesses. Notably, QueCC continues to achieve high-utilization even under extreme contention model. For example, even scaling to 32 worker threads, over the 80% of the time is dedicated to useful 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 5a, shows the throughput results of QueCC compared with other concurrency control protocols. We use a write-intensive workload



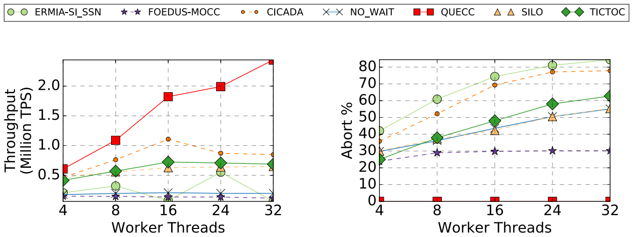
(a) Throughput **(b) Abort Percentage**

Figure 5: Variable contention (θ) on write-intensive YCSB workload



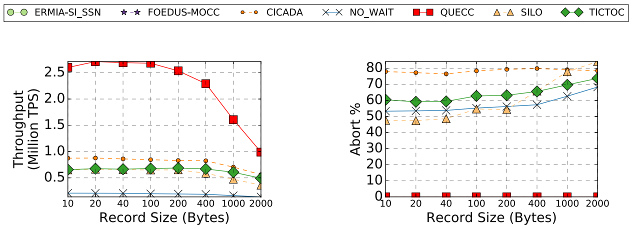
(a) Throughput **(b) Abort percentage**

Figure 7: Results for varying the percentage of write operations in each transaction. High contention, $\theta = 0.99$



(a) Throughput **(b) Abort percentage**

Figure 6: Scaling Worker Threads Under Write Intensive Workload. High contention, $\theta = 0.99$



(a) Throughput **(b) Abort Percentage**

Figure 8: Results for varying the size of records under high contention, $\theta = 0.99$.

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.

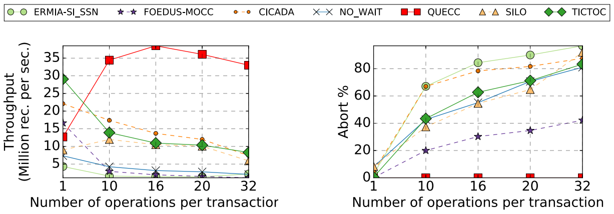
Scalability We evaluate the scalability of QueCC by varying the number of worker threads while maintaining a skewed, write-intensive access pattern. We observe that all other approaches scale poorly under highly concurrent access scenario (6a) despite employing techniques to reduce the cost of contention (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 6b shows that 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 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 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., [31]) 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%.

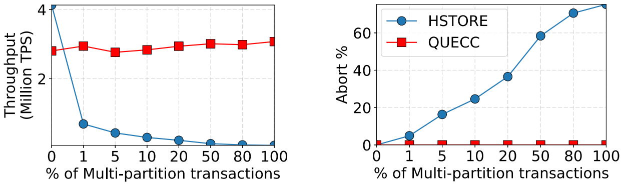
Effect of Record Sizes Having larger record sizes may also negatively affect the performance of logging component as shown in Figure 8. Since the undo log maintains a copy of every modified record, the logging throughput suffers when large records are used.



(a) Throughput

(b) Abort percentage

Figure 9: Results for varying the number of operations in each transaction. High contention, $\theta = 0.99$



(a) Throughput

(b) Abort percentage

Figure 10: Results of multi-partition transactions with comparison to H-Store.

One approach to handle the logging is to exploit the notion of "abortabity" of the transaction last updated the record, and repurpose the key principle of EWV[10].[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.

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 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 10 illustrates that QueCC throughput virtually remains the same regardless of the percentage of multi-partition transactions. 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.

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.

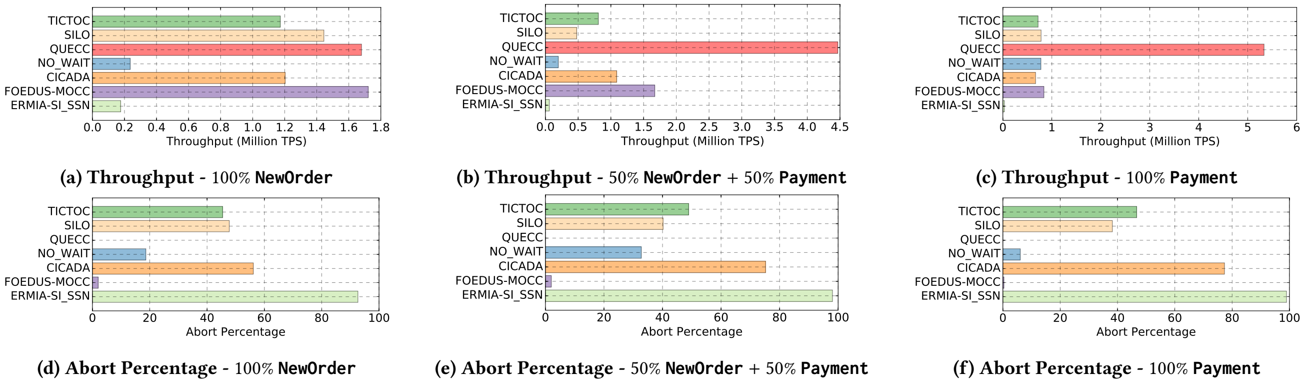
Figure 11 captures the throughput and the abort percentage. With a workload mix of 100% `Payment` transactions, Figure 11c, QueCC performs 6.34× better than the other approaches. With the a 50% `Payment` transaction mix, QueCC improves by nearly 2.7× over FOEDUS with MOCC. Despite the skewness towards 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.

6 RELATED WORK

There have been extensive research on concurrency control approaches, and there many excellent publications dedicated to this topic (e.g., [2, 12, 23, 35]). 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 paper that started to question the status quo for concurrency mechanism was H-Store [19]. H-Store imagined a simple

[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 - 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 11: Results for 32 worker threads for TPC-C benchmark. Number of warehouses = 1.

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 [28] 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[45]. Therefore, several research proposals focused on the optimistic concurrency control (OCC) approach (e.g., [31, 32, 39, 40, 46, 47]), which is originally proposed by [24]. Tu et al.'s SILO [39] 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 [46] improves concurrency by using a data-driven timestamp management protocol. Both BCC [47] and MOCC [40] 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. [25] revisited concurrency control for in-memory stores and proposed a multi-version, optimistic concurrency control with speculative reads. Sadoghi et al. [31, 32] 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 [26]. 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. [29] 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. [44] builds dependency graphs for a batch of transactions that dictates execution orders. Faleiro et al. [10] 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 [41] 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.,[9, 37]) 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[37] 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 [4] 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 [9] 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.

7 CONCLUSION

In this paper, 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.

REFERENCES

[1] A. Adya, B. Liskov, and P. O'Neil. 2000. Generalized isolation level definitions. In *Proc. ICDE*. 67–78. DOI:https://doi.org/10.1109/ICDE.2000.839388

[2] Arthur J. Bernstein, David S. Gerstl, and Philip M. Lewis. 1999. Concurrency Control for Step-decomposed Transactions. *Inf. Syst.* 24, 9 (Dec. 1999), 673–698. http://portal.acm.org/citation.cfm?id=337922

[3] Philip A. Bernstein and Nathan Goodman. 1981. Concurrency Control in Distributed Database Systems. *ACM Comput. Surv.* 13, 2 (June 1981), 185–221. DOI: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 *2012 IEEE 18th International Conference on Parallel and Distributed Systems*. 572–579.

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

[6] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. 2010. Schism: A Workload-driven Approach to Database Replication and Partitioning. *Proc. VLDB Endow.* 3, 1-2 (Sept. 2010), 48–57. DOI:https://doi.org/10.14778/1920841.1920853

[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*. ACM, 1–8. DOI:https://doi.org/10.1145/602259.602261

[8] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. 1976. The Notions of Consistency and Predicate Locks in a Database System. *Commun. ACM* 19, 11 (Nov. 1976), 624–633. DOI:https://doi.org/10.1145/360363.360369

[9] Jose M. Faleiro and Daniel J. Abadi. 2015. Rethinking Serializable Multiversion Concurrency Control. *Proc. VLDB Endow.* 8, 11 (July 2015), 1190–1201. DOI: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. DOI: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*. ACM, 15–26. DOI:https://doi.org/10.1145/2588555.2610529

[12] Jana Giceva and Mohammad Sadoghi. 2018. Hybrid OLTP and OLAP. In *Encyclopedia of Big Data Technologies*, Sherif Sakr and Albert Zomaya (Eds.). Springer International Publishing, Cham, 1–8. DOI:https://doi.org/10.1007/978-3-319-63962-8_179-1

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

[14] Suyash Gupta and Mohammad Sadoghi. 2018. EasyCommit: A Non-blocking Two-phase Commit Protocol. In *Proceedings of the 21th International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018.* 157–168. DOI:https://doi.org/10.5441/002/edbt.2018.15

[15] 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. DOI:https://doi.org/10.14778/3055540.3055548

[16] Hewlett Packard Enterprise. 2017. HPE Superdome Servers. https://www.hpe.com/us/en/servers/superdome.html. (2017).

[17] Hewlett Packard Labs. 2017. The Machine: A new kind of computer. http://labs.hpe.com/research/themachine. (2017).

[18] R. Jimenez-Peris, M. Patino-Martinez, and S. Arevalo. 2000. Deterministic scheduling for transactional multithreaded replicas. In *Proc. IEEE SRDS*. 164–173. DOI:https://doi.org/10.1109/RELDI.2000.885404

[19] 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. DOI:https://doi.org/10.14778/1454159.1454211

[20] Bettina Kemme and Gustavo Alonso. 2000. Don'T Be Lazy, Be Consistent: Postgres-R, A New Way to Implement Database Replication. In *Proc. VLDB*. Morgan Kaufmann Publishers Inc., 134–143. http://dl.acm.org/citation.cfm?id=645926.671855

[21] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2016. ERMIA: Fast Memory-Optimized Database System for Heterogeneous Workloads. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, New York, NY, USA, 1675–1687. DOI:https://doi.org/10.1145/2882903.2882905

[22] Hideaki Kimura. 2015. FOEDUS: OLTP Engine for a Thousand Cores and NVRAM. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. ACM, New York, NY, USA, 691–706. DOI:https://doi.org/10.1145/2723372.2746480

[23] Vijay Kumar (Ed.). 1995. *Performance of Concurrency Control Mechanisms in Centralized Database Systems.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[24] H. T. Kung and John T. Robinson. 1981. On Optimistic Methods for Concurrency Control. *ACM Trans. Database Syst.* 6, 2 (June 1981), 213–226. DOI: https://doi.org/10.1145/319566.319567

[25] Per A. Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. 2011. High-performance Concurrency Control Mechanisms for Main-memory Databases. *Proc. VLDB Endow.* 5, 4 (Dec. 2011), 298–309. DOI: https://doi.org/10.14778/2095686.2095689

[26] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. 2017. Cicada: Dependably Fast Multi-Core In-Memory Transactions. In *Proc. SIGMOD*. ACM, 21–35. DOI: https://doi.org/10.1145/3035918.3064015

[27] Mellanox Technologies. 2017. Multicore Processors Overview. http://www.mellanox.com/page/multi_core_overview. (2017).

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

[29] Kun Ren, Jose M. Faleiro, and Daniel J. Abadi. 2016. Design Principles for Scaling Multi-core OLTP Under High Contention. In *Proc. SIGMOD*. ACM, 1583–1598. DOI: https://doi.org/10.1145/2882903.2882958

[30] 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. DOI: https://doi.org/10.14778/2732951.2732955

[31] Mohammad Sadoghi, Souvik Bhattacherjee, Bishwaranjan Bhattacharjee, and Mustafa Canim. 2018. 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.* 540–551. DOI: https://doi.org/10.5441/002/edbt.2018.65

[32] Mohammad Sadoghi, Mustafa Canim, Bishwaranjan Bhattacharjee, Fabian Nagel, and Kenneth A. Ross. 2014. Reducing Database Locking Contention Through Multi-version Concurrency. *Proc. VLDB Endow.* 7, 13 (Aug. 2014), 1331–1342. DOI: https://doi.org/10.14778/2733004.2733006

[33] Sgi. 2017. SGI UV 3000 and SGI UV 30. https://www.sgi.com/products/servers/uv/uv_3000_30.html. (2017).

[34] Dennis Shasha, Francois Llirbat, Eric Simon, and Patrick Valduriez. 1995. Transaction Chopping: Algorithms and Performance Studies. *ACM Trans. Database Syst.* 20, 3 (Sept. 1995), 325–363. DOI: https://doi.org/10.1145/211414.211427

[35] Alexander Thomasian. 1998. Concurrency Control: Methods, Performance, and Analysis. *ACM Comput. Surv.* 30, 1 (March 1998), 70–119. DOI: https://doi.org/10.1145/274440.274443

[36] Alexander Thomson and Daniel J. Abadi. 2015. CalvinFS: Consistent WAN Replication and Scalable Metadata Management for Distributed File Systems. In *Proc. FAST*. USENIX Association, 1–14. http://portal.acm.org/citation.cfm?id=2750483

[37] 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*. ACM, 1–12. DOI: https://doi.org/10.1145/2213836.2213838

[38] TPCC. TPC-C, On-line Transaction Processing Benchmark. (????). http://www.tpc.org/tpcc/.

[39] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-memory Databases. In *SOSP*. ACM, 18–32. DOI: https://doi.org/10.1145/2517349.2522713

[40] Tianzheng Wang and Hideaki Kimura. 2016. Mostly-optimistic Concurrency Control for Highly Contended Dynamic Workloads on a Thousand Cores. *Proc. VLDB Endow.* 10, 2 (Oct. 2016), 49–60. DOI: https://doi.org/10.14778/3015274.3015276

[41] Zhaoguo Wang, Shuai Mu, Yang Cui, Han Yi, Haibo Chen, and Jinyang Li. 2016. Scaling Multicore Databases via Constrained Parallel Execution. In *Proc. SIGMOD*. ACM, 1643–1658. DOI: https://doi.org/10.1145/2882903.2882934

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

[43] Arthur T. Whitney, Dennis Shasha, and Stevan Apter. 1997. High Volume Transaction Processing without Concurrency Control, Two Phase Commit, Sql or C++. In *HPTS*.

[44] C. Yao, D. Agrawal, G. Chen, Q. Lin, B. C. Ooi, W. F. Wong, and M. Zhang. 2016. Exploiting Single-Threaded Model in Multi-Core In-Memory Systems. *IEEE TKDE* 28, 10 (2016), 2635–2650. DOI: https://doi.org/10.1109/TKDE.2016.2578319

[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. DOI: https://doi.org/10.14778/2735508.2735511

[46] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. 2016. TicToc: Time Traveling Optimistic Concurrency Control. In *Proc. SIGMOD*. ACM, 1629–1642. DOI: https://doi.org/10.1145/2882903.2882935

[47] Yuan Yuan, Kaibo Wang, Rubao Lee, Xiaoning Ding, Jing Xing, Spyros Blanas, and Xiaodong Zhang. 2016. BCC: Reducing False Aborts in Optimistic Concurrency Control with Low Cost for In-memory Databases. *Proc. VLDB Endow.* 9, 6 (Jan. 2016), 504–515. DOI: https://doi.org/10.14778/2904121.2904126