Dong ZY, Tang CZ, Wang JC *et al.* Optimistic transaction processing in deterministic database. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 35(2): 382–394 Mar. 2020. DOI 10.1007/s11390-020-9700-5

Optimistic Transaction Processing in Deterministic Database

Zhi-Yuan Dong, Chu-Zhe Tang, Jia-Chen Wang, Zhao-Guo Wang*
Hai-Bo Chen, *Distinguished Member*, *CCF*, *Senior Member*, *ACM*, *IEEE*, and
Bin-Yu Zang, *Distinguished Member*, *CCF*, *Member*, *ACM*

Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, Shanghai 200240, China

E-mail: dongzy829@gmail.com; {t.chuzhe, wangjc, zhaoguowang, haibochen, byzang}@sjtu.edu.cn

Received May 8, 2019; revised September 2, 2019.

Abstract Deterministic databases can improve the performance of distributed workload by eliminating the distributed commit protocol and reducing the contention cost. Unfortunately, the current deterministic scheme does not consider the performance scalability within a single machine. In this paper, we describe a scalable deterministic concurrency control, Deterministic and Optimistic Concurrency Control (DOCC), which is able to scale the performance both within a single node and across multiple nodes. The performance improvement comes from enforcing the determinism lazily and avoiding read-only transaction blocking the execution. The evaluation shows that DOCC achieves 8x performance improvement than the popular deterministic database system, Calvin.

Keywords deterministic database, concurrency control, scalability

1 Introduction

Deterministic databases [1–3] execute transactions in a predetermined order. This makes the systems can eschew the distributed protocol which is a major performance factor of distributed databases systems, and have promising performance even under high contention workload as all transactions are ordered in advance.

Calvin [1] is the most popular deterministic database and dominates most deterministic database systems design [3–5]. It first puts all received transactions in a serial order. Then, each machine executes the transactions following the predetermined order.

To enforce the order, each machine has a scheduler thread to constrain the transaction execution. It grants the locks to requesting transactions in the strict predetermined order. As a result, the transactions will deterministically access the data. However, since the scheduler thread grants the locks sequentially, this will serialize the execution of all conflicting transactions even if there are parallelism opportunities. For example, considering transaction T_1 updates A and C, while transaction T_2 updates B and C, even they have conflicting access on C, they still can concurrently access records A and B. But, Calvin will first grant the locks of both A and C to T_1, and T_2 cannot start execution until T_1 commits.

To address the scalability problem in Calvin, we propose a new protocol, called Deterministic and Optimistic Concurrency Control (DOCC). It can unleash potential parallelism opportunities with the deterministic guarantee. DOCC's key to solving the scalability issue in Calvin is to enforce the predetermined order lazily. In particular, DOCC executes the transactions concurrently and constrains the validation order of dependent transactions. In the above example, under DOCC, T_1 and T_2 can access records of A and B concurrently. However, T_2 cannot perform validation until T_1 commits. Furthermore, DOCC can also avoid read-only transaction blocking execution with snapshots.

Meanwhile, there are also some studies targeting to improve the scalability of deterministic databases, like Lazy Evaluation [3], VLL [4] and BOHM [5]. However, previous work (including Calvin) must either know the read/write sets or statically analyze the transactions in advance. Relatively, DOCC does not need any pre-

Regular Paper
Special Section of ChinaSys 2019
*Corresponding Author

knowledge of transactions due to optimistic execution and lazy scheduling, which improves the generality of deterministic databases.

The following are our contributions.

• We propose DOCC. It improves Calvin's scalability by unleashing potential parallelism opportunities.

• We propose an optimization which prevents read-only transactions from blocking the execution.

• We provide a retry strategy which can reuse previous execution results.

• We implement a prototype of DOCC and the experiment shows DOCC can outperform Calvin with 4.31x–8.46x in TPC-C benchmark.

2 Motivation and Approach

Deterministic database is able to eliminate the distributed commit protocol and reduce the contention in the workload by executing the transactions in a predetermined order [1]. However, it also has drawbacks that its performance cannot scale up in a single machine, and the supported transaction types are restricted. In this section, we will describe the problems of the deterministic database in detail by studying a popular deterministic database system, Calvin. Then, we exploit the potential opportunities and ideas for improvement.

2.1 Calvin and Its Scalability Issues

Calvin is a layered system. In the sequencing layer, it generates a dependency graph according to the global arriving order of the received transactions. In Calvin, the sequencing service is distributed for scalability, which is composed of multiple sequencers. Then, each sequencer broadcasts the transactions and dependency graph to the scheduler on each machine (scheduling layer). Each scheduler aggregates the requests from the sequencers and enforces the transactions to execute in the dependent order determined in the sequencing layer.

Before demonstrating the issues in Calvin, we need to describe the scheduling layer in more details. Each machine has a scheduling thread which is used to arrange the transactions' execution order by granting the locks deterministically. Specifically, the scheduler analyzes the read/write set of all received transactions, and then grants the locks to transactions strictly in the order determined by the sequencer. Lastly, it assigns these transactions to multiple worker threads for parallel execution.

Fig.1(a) gives an example, transaction T_1 reads records A and C, while transaction T_2 updates B and C. Consider Calvin receives these two transactions concurrently and serializes T_1 before T_2 at the sequencer layer. After the scheduler receives these two transactions from the sequencer, it first analyzes T_1's

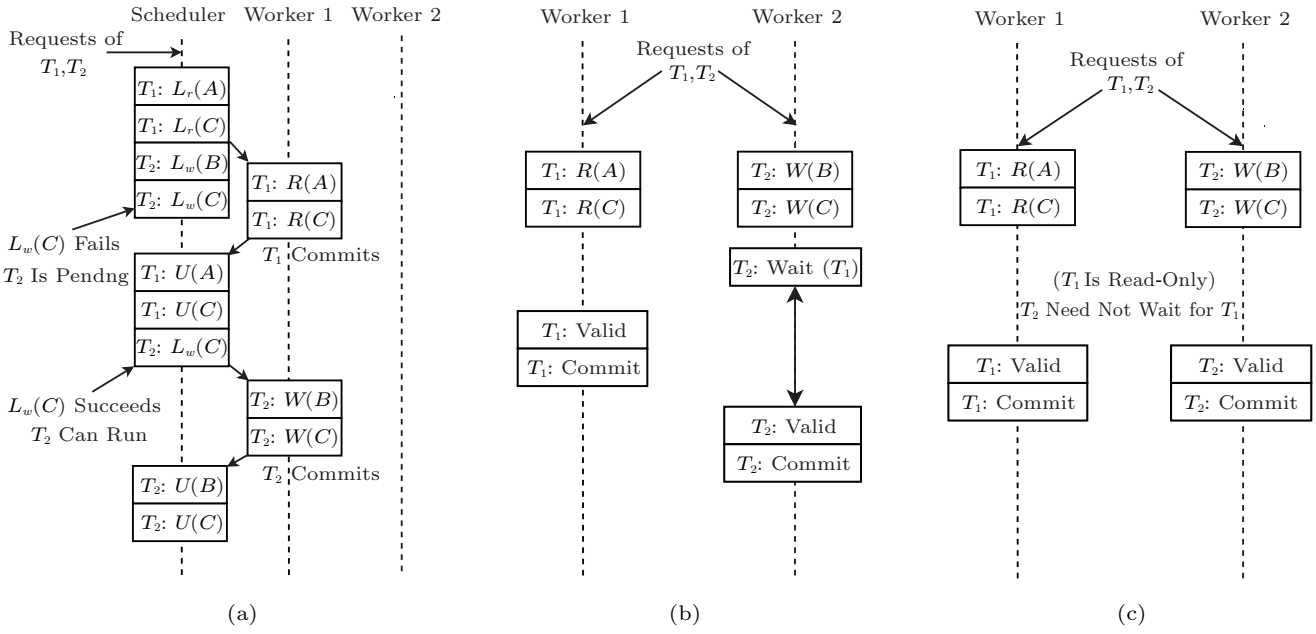


Fig.1. (a) Example of Calvin's deterministic locking. (b) Example of DOCC. (c) Example of DOCC with snapshot. L_r, L_w, U, R, and W mean "acquire read lock", "lock write lock ", "unlock", "read", and "write" respectively.

read/write sets and grants locks of A and C to T_1. When it tries to grant C's lock to T_2, it will find the lock has been already held by T_1 and suspend T_2 until T_1 commits.

Fig.2 shows that the scheduler on each machine limits the performance scalability within the single node. We run TPC-C benchmark with Calvin and evaluate the performance of the single node in the cluster. The performance cannot scale after four cores because of the sequential scheduling strategy.

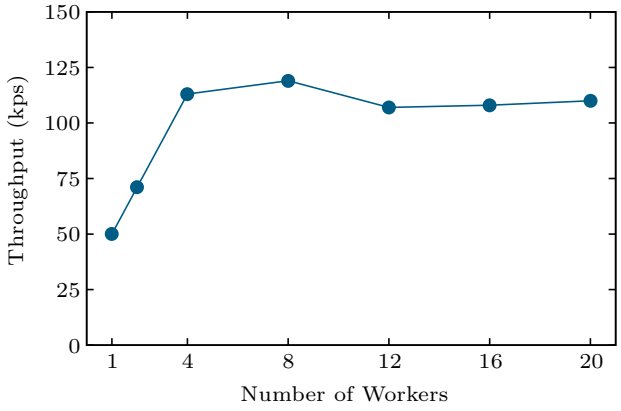


Fig.2. Calvin's multi-core scalability on 24-core machine in TPC-C standard-mix benchmark. Each worker thread is dedicated to one core. Four cores are responsible for sequencing and scheduling transactions.

2.2 DOCC's Approach

When using Calvin's deterministic scheduling, many opportunities for parallelism are lost even for conflicting transactions. Considering the above example, T_1 and T_2 can access A and B concurrently, and only need to be serialized on the access of C. However, Calvin will make these two transactions' execution sequential. To exploit the potential parallelism, we propose a new deterministic concurrency control, DOCC, which has scalable performance in a single node or across multiple nodes. Below, we discuss the main idea of DOCC.

Enforcing the Deterministic Order Lazily. After receiving transactions from the sequencers, DOCC directly executes all transactions concurrently and validates the execution results in a deterministic manner. To be more specific, each record is associated with a version number which indicates the last transaction which updates it. During the execution, the transaction reads the record content and its version into its local read set, and buffers the updates in its write set. Then, the transaction needs to wait for all its dependent transactions decided by the sequencer to commit before the validation. DOCC performs the validation by comparing the current version with the version number buffered in the read set. Last, it will commit all its updates in the write set. Fig.1(b) shows how DOCC arranges the parallel execution of T_1 and T_2 in Fig.1(a). After receiving T_1 and T_2 from the sequencer, the scheduler directly executes T_1 and T_2 concurrently. During the execution, T_1 and T_2 buffer the record version or the updates in their read/write sets. Since T_1 is serialized before T_2 by the sequencer, DOCC will block T_2's validation phase until T_1 finishes.

Avoiding Read-Only Transactions Blocking the Execution. For the above algorithms, a transaction T cannot perform validation until all dependent transactions commit. This limits the scalability. However, if T is serialized after a read-only transaction T_r, then it is not necessary to wait for T_r to commit before its validation. Based on this observation, DOCC leverages the multiple version design to avoid read-only transaction blocking the execution. When a transaction commits, it creates a new version for the updated record instead of in-place update. This can avoid overwriting the content of records which may be still needed by previous read-only transactions. Considering the above example, T_1 is a read-only transaction which cannot invalidate T_2's execution. Thus T_2 does not need to wait for T_1's finish. When T_2 commits, it creates a new version on both records B and C, since T_1 still needs the old version.

3 DOCC

This section describes how transactions are executed in DOCC. The general idea is to execute transactions optimistically and use a sequential validation mechanism to guarantee the determinism — the execution with DOCC should be serializable to the serial execution in the predetermined order. In the rest of the section, we will first present the record layout and the basic algorithm of DOCC (Subsection 3.1). Then we introduce two novel optimizations: 1) eliminating unnecessary waits with snapshots (Subsection 3.2) and 2) speeding up retry with data prefetching (Subsection 3.3), and present how we efficiently perform garbage collection to support these two optimizations (Subsection 3.4). Then we prove the correctness of DOCC (Subsection 3.5). Lastly but certainly not least we discuss the difference between DOCC and existing concurrency controls and why DOCC can improve the generality of deterministic databases (Subsection 3.6).

3.1 Basic Algorithm

The basic algorithm (Algorithm 1) of DOCC includes four phases and bears resemblance to original optimistic concurrency control (OCC) protocol except for the waiting phase that guarantees the serializability to a predetermined order. The four phases are as follows.

- Execution phase executes transactions optimisticly and tracks necessary record metadata.
- Waiting phase stalls transaction validation until its dependent transaction has committed.
- Validation phase validates the execution result and aborts the transaction upon the failure of validation.
- Commit phase commits the execution result and sets the transaction state to Committed.

Algorithm 1. DOCC

```
   Input: T_N, T_N-dep
1  Execution Phase
2      foreach key in T_N do
3          record ← read(key)
4          if read then
5              buf ← record.data
6          else
7              buf ← new data
8          version ← record.version
9  Waiting Phase
10     while T_N-dep.state != Committed do
11         Spin
12 Validation Phase
13     foreach record, version in read set do
14         if version != record.version then
15             Abort and directly retry T_N;
16 Commit Phase
17     foreach record, buf in write set do
18         new_record.data ← buf
19         new_record.version ← N
20         write(new_record)
21     T_N.state ← Committed
```

Before going further into detail, we briefly introduce the record layout and notations used throughout this section. In DOCC , a record consists of 1) a current version number and 2) a data pointer to the actual data. We use T_N to denote the N-th transaction in the predetermined order, and $T_{N\text{-}dep}$ to denote T_N's dependent transaction. T_N's validation phase should wait until $T_{N\text{-}dep}$ has committed to that the execution of the two transactions follows the predetermined order. Usually $T_{N\text{-}dep}$ equals T_{N-1}, but with snapshots (Subsection 3.2) $T_{N\text{-}dep}$ can be an earlier transaction in the predetermined order. For simplicity, we use the same N as the transaction ID of T_N.

Execution Phase. During the execution, for each operation within T_N, Algorithm 1 first fetches the record by the corresponding key (line 3). Then, according to the operation type, either the original data or the new data is stored into the local buffer (lines 4–7). Finally, the version number of this record is stored for future validation (line 8).

Waiting Phase. After T_N's execution phase, Algorithm 1 waits until the dependent transaction $T_{N\text{-}dep}$ has been committed before going to the validation phase (lines 10 and 11). Therefore, T_N is able to observe the latest updates from the previous transactions in its validation phase.

Validation Phase. Algorithm 1 validates that T_N has read all the latest records by comparing the version numbers stored during the execution phase with the record's current version number (line 14). When the validation fails, Algorithm 1 aborts and retries T_N immediately (line 15). To ensure determinism, when T_N is aborted, subsequent transactions should keep on waiting until the retry has successfully committed. The details about abort and retry will be described in Subsection 3.3.

Commit Phase. Algorithm 1 installs the buffered updates of T_N to the actual records with new version numbers (lines 17–20) and marks the transaction state as Committed (line 21). After T_N has been committed, its subsequent transactions can then proceed to the validation phase. Since the waiting phase guarantees there is only one transaction running the validation phase and the commit phase at a time, Algorithm 1 does not acquire any write locks.

3.2 Eliminating Unnecessary Waits with Snapshots

To eliminate unnecessary waits while respecting the predetermined order, we introduce snapshots to DOCC (Algorithm 2). In Algorithm 1, assuming $T_{N\text{-}dep}$ is a read-only transaction, T_N always spins in the waiting phase until $T_{N\text{-}dep}$ has committed, even if $T_{N\text{-}dep}$ does not update any records that T_N needs. This presents an opportunity for further optimizing DOCC performance by eliminating such unnecessary waits. However, we cannot simply skip the waiting phase of T_N because then T_N might commit during $T_{N\text{-}dep}$'s execution phase and accidentally overwrite the records with a newer ver-

sion that $T_{N\text{-}dep}$ should not read, thereby the determinism is violated.

Algorithm 2. DOCC with Snapshot

```
   Input: T_N, T_{N-dep}
1  Execution Phase
2      foreach key in T_N do
3          record ← snapshot_read(key, N)
4          if read then
5              buf ← record.data
6          else
7              buf ← new data
8          version ← record.version
9  Waiting Phase // Same as Algorithm 1
10 Validation Phase
11     foreach record, version in read set do
12         record ← snapshot_read(record, N)
13         if version != record.version then
14             Abort and directly retry T_N
15 Commit Phase
16     foreach old_record, buf in write set do
17         new_record.data ← buf
18         new_record.prev ← old_record
19         new_record.version ← N
20         write(new_record)
21     T_N.state ← Committed
```

To execute transactions with snapshots, we keep the current version of records and previous versions that would have been overwritten in Algorithm 1. We further include a *prev* pointer in the record layout, which points to the record's latest version, so that all the versions are chained together (line 18). Garbage collection for the stale versions will be discussed in Subsection 3.4.

To eliminate unnecessary waits, we set T_N's dependent transaction $T_{N\text{-}dep}$ to the last read-write transaction prior to T_N, rather than T_{N-1}. In this way, read-only transactions will not block subsequent transactions while read-write transactions can still observe the latest updates from previous read-write transactions during the validation phase (line 10).

To ensure determinism, transaction T_N will read the latest version of records visible to T_N from existing snapshots (line 3). During the validation phase, T_N will reread the snapshots to ensure that the versions T_N has read are still the latest ones (line 12).

Fig.3 illustrates how this optimization works. In this example, we have three transactions T_1, T_2, T_3 and the record R with initial value R_0. T_1 and T_3 read and update record R to R_1 and R_3 respectively while T_2 only reads record R. Assuming we have two worker threads, in the beginning, T_1 and T_2 will both enter the execution phase and read R_0. Since T_2 depends on T_1, it will wait for T_1 to commit and set record R to R_1 before entering the validation phase. During the validation phase, T_2 will find that it should read the value R_1 instead of R_0; therefore Algorithm 2 will abort and retry T_2. At the same time T_3 starts to execute and commit after setting record R to R_3. Since we have snapshots to track old versions of records, the execution of T_3 will not affect the retry of T_2.

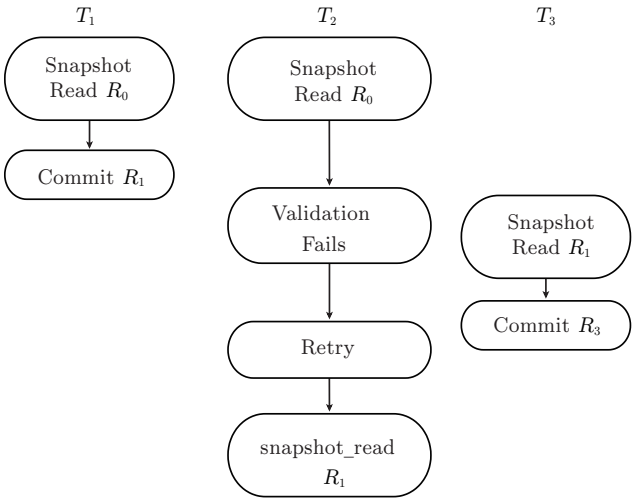


Fig.3. Example showing how multi-version snapshot works for read-only transactions.

3.3 Speeding up Retry with Data Prefetching

To speed up transaction retry and minimize the blocking cost, we introduce an indirection-based data prefetching mechanism. In DOCC, if transaction T fails during the validation phase, it will be aborted and retried immediately. Such retry hurts DOCC performance significantly since any transactions that depend on T, either directly or indirectly, will have to wait until T has successfully committed for determinism. However, in many online transactional processing (OLTP) workloads, transaction execution time is largely spent on data accessing, e.g., traversing index structures. Based on this observation, we propose to use another layer of indirection to prefetch data during retry, effectively eliminating most of the data accessing cost.

To eliminate data accessing cost, we need to be able to access the records directly during retry. If we simply track the addresses of records during the first aborted execution, we can easily run into segmentation faults since records might be deleted at retry time. Moreover, to be able to find the latest version from a potentially old record's address, we need *next* pointers in the record

layout and perform a time-consuming traversal, due to the number of random memory accesses.

To address these issues, we propose 1) to use logical deletion with special empty records and 2) to index another data structure called record placeholders, which contains nothing but a pointer to the latest record, instead of directly indexing the records. As Fig.4 shows, all versions of the same records, including versions for logical deletion, are linked after the record placeholder and the record placeholder will not be deleted directly by any other transactions. In this way, during retry, transactions can directly access the latest version of records, by accessing the record placeholders gathered in the first aborted execution. The actual deletion operation is performed by garbage collection (Subsection 3.4).

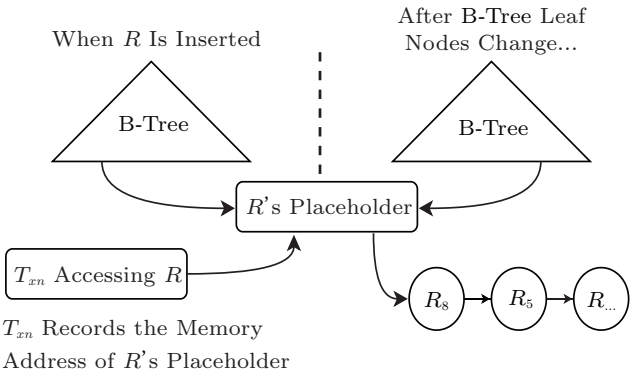


Fig.4. Example of placeholder. T_{xn} means a transaction.

Algorithm 3 is a simplified algorithm for transaction retry in DOCC with data prefetching optimization. When retrying a transaction T, we do not need the waiting phase or the validation phase anymore, since all the transactions that can update records have already committed before retry. Therefore T is able to observe the latest updates.

Algorithm 3. DOCC's Retry

```
   Input: T_N
1  Execution Phase
2      foreach key, placeholder in T_N do
3          if key first appears in retry then
4              record ← snapshot_read(key, N)
5          else
6              record ←
                 placeholder_read(placeholder, N)
7          buf ← record.data
8          version ← record.version
9  Commit Phase // Same as Algorithm 2
```

3.4 Garbage Collection

By introducing snapshots and logical deletes, garbage collection (GC) is required to reclaim memory occupied by out-of-date records. In DOCC, garbage collection is divided into two parts: 1) trimming the version chain of each record and 2) removing unused record placeholders. Meanwhile, a background thread is dedicated for GC.

3.4.1 Trimming the Version Chain of Each Record

To trim off the old versions that will never be used again, DOCC maintains the smallest transaction ID among current running transactions, TID_{earlist}. Each time when the GC thread tries to trim the record's version chain, it starts by finding the first version R_i whose version ID is smaller than or equal to TID_{earlist} as the reclamation starting point, and then the GC thread recursively reclaims memory starting from version R_i's *prev* pointer.

Fig.5 illustrates how this GC procedure takes action. Considering that we have record R with three versions R_1, R_2 and R_4 and currently TID_{earlist} is 3. The GC thread runs in the background scanning the version chain of the record R and finds that the starting point of reclamation is R_2. Then the GC thread will start reclaiming memory from R_2's *prev* pointer and therefore the stale version R_1 is reclaimed. It is important that GC thread leaves the reclamation starting point (R_2 in this example) intact since that version is still visible to running transaction (T_3 in this example).

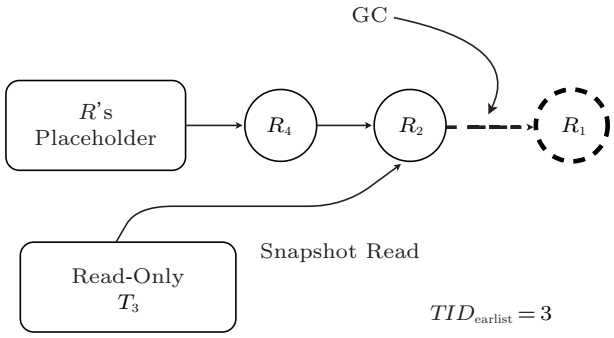


Fig. 5. Example explaining how record versions are garbage-collected.

3.4.2 Removing Unused Record Placeholders

The GC procedure described above can effectively remove stale versions, but it fails to remove the unused record placeholders, whose latest version is a logical delete, from the index structure, since it will preserve

at least one version of the record, the reclamation starting point. On the other hand, it is necessary to remove these record placeholders as well; otherwise the system will eventually contain too many logical deleted records and suffer from severe performance degradation.

We cannot directly reclaim unused record placeholders as any running transaction can potentially be accessing them and concurrent reclamation will, therefore, cause invalid memory accesses. One way to safely remove them is to add a reference counter to each record placeholder and reclaim memory when there is no access. However, this approach introduces a significant amount of synchronization overhead [6].

We use a read-copy-update (RCU) style reclamation mechanism [7–10] to safely remove unused record placeholders while maintaining good performance. For each unused record placeholder, we mark it as "sealed" along with TID_{GC}, the largest transaction ID among current running transactions. These record placeholders will no longer be modified and will be reclaimed after transaction $T_{TID_{\text{GC}}}$ has committed. If a running transaction attempts to update a new version of the record, it will create a new record placeholder and store the new version as the latest one. Our index structure makes sure that the "sealed" record placeholders do not conflict with new ones.

3.5 Correctness

In this subsection, we present a proof sketch of the correctness that DOCC satisfies deterministic scheduling. Given transactions T_1–T_N, the execution with DOCC is serializable to executing T_1–T_N serially, denoted by $T_1 \rightarrow T_2 \rightarrow ... \rightarrow T_N$ in the serialization graph.

We prove this by contradiction. We assume there are two transactions T_i and T_j $(i < j)$, and they conflict with each other. We also assume $T_j \rightarrow T_i$, i.e., T_i commits after T_j.

If T_i is a read-write transaction, note that DOCC ensures that 1) if T_i is validating or committing, T_i's subsequent transactions are uncommitted; 2) uncommitted transaction's updates are invisible to other transactions. If T_i is a read-only transaction, snapshot ensures that T_i uses i as a version to read a consistent snapshot whose version is smaller than i. Thus T_i will never observe T_j's update or commit after T_j.

At the same time, DOCC also guarantees that if T_j is validating and committing, T_1–T_{j-1} have all committed their updates. Thus T_j must observe T_i's updates and commit after T_i.

Therefore $T_j \rightarrow T_i$ cannot exist, and hence DOCC guarantees the execution is serializable to execute T_1–T_N serially.

3.6 Discussion

Comparing DOCC with Optimistic Concurrency Control (OCC) & Multi-Version Concurrency Control (MVCC). Inspired by OCC, DOCC allows transactions to execute optimistically without coordination and lazily enforces the correctness. Inspired by MVCC, DOCC prevents read-only transactions from blocking execution. Comparing these concurrency controls, the major differences are as follows.

- DOCC must follow the predetermined order in deterministic databases and has an extra waiting phase to guarantee that only one transaction can enter the validation phase and the commit phase on each node.
- DOCC does not need to acquire locks for write set as there is only one writer in the validation phase and the commit phase each time.
- DOCC directly uses the transaction ID which represents the predetermined order as the version number (or timestamp), avoiding costly or complex version number management.

Improving the Generality. For existing deterministic databases [1, 3–5], they schedule transactions before execution; therefore they must know the read/write sets or statically analyze the transactions in advance. Relatively, like OCC, DOCC does not require the preknowledge of read/write sets as it can determine the read/write sets during execution. In addition, DOCC does not involve any static analysis to chop the transactions. Therefore, DOCC can relax the transaction constraint of previous work and improve the generality of deterministic databases.

4 Evaluation

In this section, we want to answer the following questions with our evaluation.

- Can DOCC's single-node performance outperform Calvin's and non-deterministic database's?
- Can DOCC's multi-node performance outperform Calvin's and non-deterministic database's?
- How much does DOCC's optimization improve performance?

4.1 Experimental Setup

Testbed. We use a rack-scale cluster with eight machines for all of our experiments. Each machine

is equipped with two 12-core Intel Xeon E5-2650 v4 processors and 128 GB of DRAM. Each machine runs Ubuntu 16.04.

Systems. As Calvin's source code① is not well optimized, we optimize Calvin with several optimizations as our baseline②. The optimizations include but are not limited to applying fast in-memory index structure, eliminating messaging serialization overhead and using encoded eight-byte word instead of a string as key.

In the following subsections (Subsection 4.1–Subsection 4.3), we refer Calvin to our optimized version of Calvin. For an apple-to-apple comparison, we then implement DOCC atop of Calvin, thereby DOCC and Calvin are based on the same codebase.

In the following experiments, we show the performance comparison among four databases.

• Calvin: highly optimized version of Calvin.

• DOCC: DOCC with snapshot and prefetching optimizations by default.

• DrTM+R: a high-performance non-deterministic database [11] based on OCC without logging enabled.

• Lazy evaluation: a high-performance deterministic database executing transactions according to dependency graph [3].

For the above four systems, the worker threads are pinned to one core. We do not use networked clients. Instead, there are dedicated transaction generators on each node issuing transactions.

As lazy evaluation is a single-node database, we cannot evaluate its scalability in distributed setting (Subsection 4.3).

TPC-C Benchmark. The TPC-C benchmark simulates a warehouse-centric eCommerce order processing system. TPC-C benchmark consists of five different transactions. Our implementation directly follows TPC-C's specification③. By partitioning warehouses, transactions in TPC-C can be changed into one-shot transactions [12].

Contention Levels. Two contention levels are used in following evaluations.

• *Low Contention.* The number of warehouses maintained on each node is equal to the number of worker threads per node. The warehouses are accessed evenly.

• *High Contention.* Only one warehouse is maintained on each node. The warehouses are accessed evenly.

4.2 Single-Node Performance

Because deterministic databases use four cores to sequence and schedule transactions, we use at most 20 workers per machine for all four systems for single-node evaluation. The comparative relation of four databases' performance will not change if we add four more workers to DrTM+R.

Low Contention. Fig.6 shows the TPC-C benchmark performance of four databases under low contention.

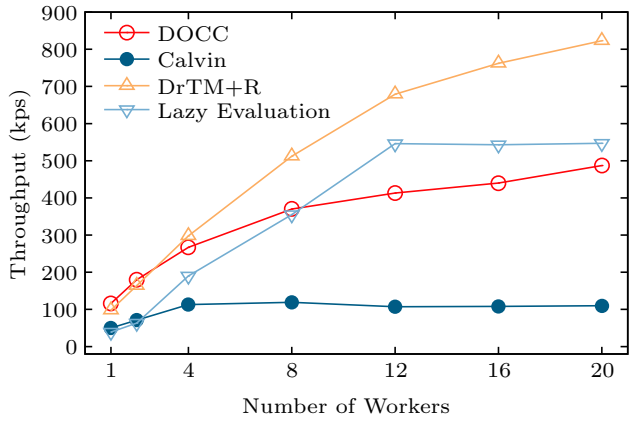


Fig.6. Single-node throughput of TPC-C benchmark under low contention.

When there is only one worker, Calvin suffers from unnecessary locking overhead, as the execution of transactions and locking are serial. Meanwhile, DOCC and DrTM+R handle transactions analogously in a single worker. Therefore, DOCC's single-thread throughput is 132% higher than Calvin's and similar to DrTM+R's.

Since Calvin's throughput is limited by the deterministic locking, the maximum throughput of Calvin is 119 kps (1 000 transactions per second) when eight workers are available. With the number of workers increasing to 20, DOCC reaches its throughput at 487 kps, which outperforms Calvin's by 4.43x.

There are two reasons why lazy evaluation has better throughput (547 kps) than DOCC. First, it lets transactions accessing the same data execute on the same CPU and thus leverages cache locality. Second, its code is highly optimized for the TPC-C benchmark. For example, it manually batches data accesses in TPC-C and only schedules them once. Lazy evaluation dedicates one thread to generate the dependency graph and

①Ren K. Calvin GitHub repository. https://github.com/yaledb/calvin, Sept. 2019.

②Evaluation not presented in the paper shows that the optimized Calvin has better performance than Calvin's source code.

③The Transaction Processing Council. TPC-C Benchmark (Revision 5.11.0). http://www.tpc.org/tpcc/, Sept. 2019.

is bottlenecked when there are 12 workers.

DrTM+R has the best peak throughput at 823 kps, as OCC favors transaction workload under low contention.

For determinism, DOCC's transactions have to enter the validation phase and the commit phase serially which may become the bottleneck. However, with snapshot and prefetching optimizations, the overhead of these two phases is much lower than that of deterministic locking.

High Contention. Fig.7 shows the TPC-C benchmark performance of four databases under high contention.

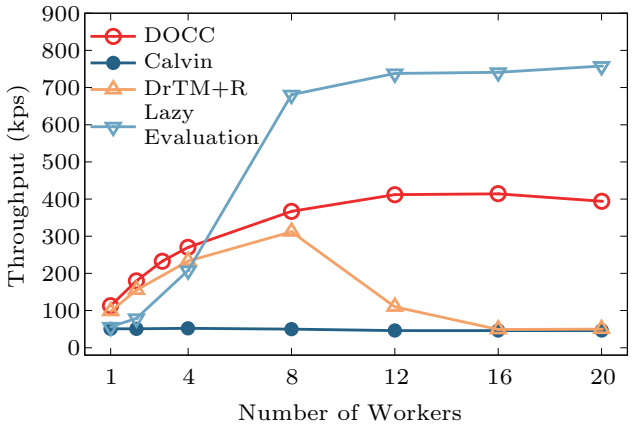


Fig.7. Single-node throughput of TPC-C benchmark under high contention.

Calvin's throughput is constant as its single-thread performance, about 50 kps. In TPC-C, payment transaction updates the warehouse table, which causes high contention when acquiring locks. Therefore, the execution of Calvin under high contention is almost serial. In addition, the locking overhead makes the performance of Calvin even worse. For DOCC, although DOCC frequently retries under high contention, prefetching used at transactions' retry makes the overhead of fetching records from storage negligible. Therefore, we can see that DOCC's peak throughput is about 412 kps, which is 8.24x higher than that of Calvin. DOCC's optimistic execution helps improve performance even under high contention.

Lazy evaluation has the best throughput at 758 kps, which is even higher than its own throughput under low contention. This is because only a little amount of data is accessed under high-contention and lazy evaluation's locality-friendly design can fully exploit the performance of cache. DrTM+R suffers from the overhead of frequent retry; therefore its peak throughput is only 312 kps with eight workers and drops dramatically when adding more workers.

4.3 Scalability

In this subsection, we use an eight-machine cluster to evaluate the scalability. On each machine, 20 and 24 workers are used for deterministic databases and DrTM+R, respectively.

Fig.8 and Fig.9 show the scalability of three databases with TPC-C benchmark. The per-node throughput of two deterministic databases is roughly the same as their single-node throughput and the scalability of both deterministic databases is nearly linear. At the point of eight machines, DOCC outperforms Calvin 4.31x under low contention and 8.46x under high contention, as DOCC's single-node performance is higher.

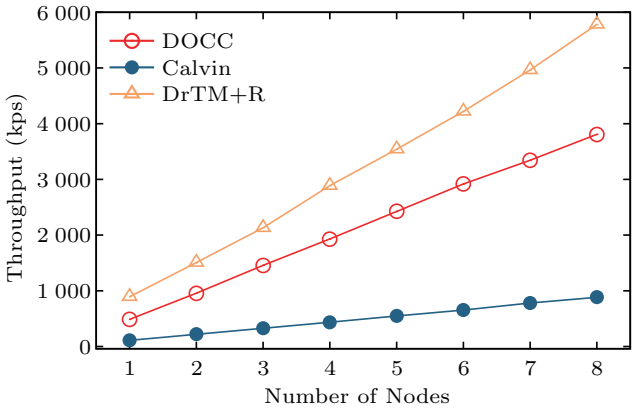


Fig.8. Scalability of TPC-C benchmark under low contention.

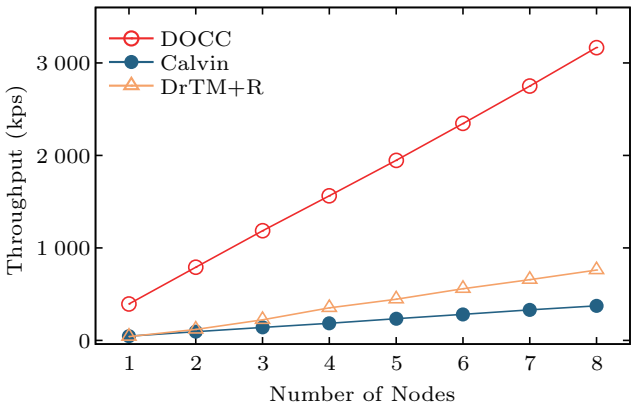


Fig.9. Scalability of TPC-C benchmark under high contention.

By executing transactions according to the predetermined order, each node of the deterministic database can decide whether to commit the transaction individually instead of applying distributed commit proto-

col. Therefore, both deterministic databases have good scalability across multiple nodes.

DrTM+R is a well-designed database, as the TPC-C benchmark partitioned by warehouses is a scalable workload. DrTM+R can also have good scalability. The comparative relation of DOCC's and DrTM+R's performance is the same as that of single-node evaluation (Subsection 4.2).

4.4 Factor Analysis

To understand the benefits of DOCC's optimizations, we show a factor analysis by running TPC-C benchmark under low contention. We use a single machine with 20 workers to do the analysis. The result is in Fig.10. "Basic" represents the basic algorithm of DOCC (Subsection 3.1). When adding snapshot ("+Snapshot") (Subsection 3.2), the performance achieves 2.05x than that of "Basic". After prefetching ("+Prefetching") (Subsection 3.3) is applied cumulatively, the final performance achieves 2.66x. The analysis demonstrates that both optimizations of DOCC have significant improvements towards performance, because snapshot eliminates the blocking of waiting for previous read-only transactions and prefetching shortens the overhead of transaction's retry.

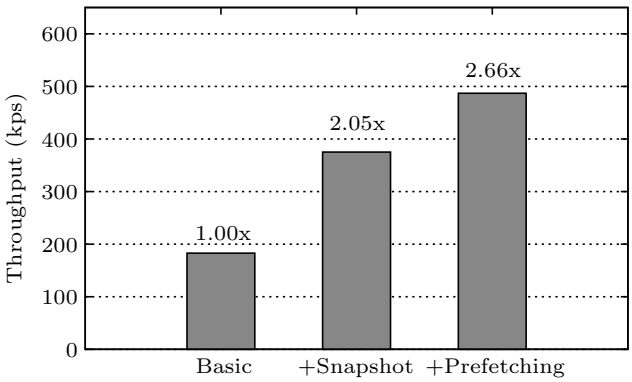


Fig.10. Factor analysis of each DOCC's optimization.

5 Related Work

Deterministic Database. Faleiro *et al.*[3] proposed a deterministic database with lazy evaluation. Given predetermined order, the transactions can be executed lazily by constructing the dependency graph. In order to determine the transaction's commit decision and read/write sets, transactions are divided into Eager and Lazy parts, which requires detailed analysis for stored procedures. DOCC does not need such analysis.

Very lightweight locking (VLL)[4] and BOHM[5] also target on improving deterministic database's scalability within one machine. VLL uses a multi-threaded lightweight locking mechanism while BOHM leverages multi-versioned storage and intra-transaction parallelism. However, in these approaches, the read/write sets or write set must be known in advance. Relatively, DOCC does not assume pre-knowledge of transactions' read/write sets.

Optimistic Concurrency Control (OCC). OCC can reduce transactions' contention and exploit transactions' parallelism thus being favoured by many databases such as FOEDUS[13], Silo[7], DrTM+R[11], Fasst[14], and DrTM+H[15]. After OCC was proposed by Kung and Robinson[16], there are many researches focusing on its variants[17–21]. Our work is also a variant of OCC which targets deterministic databases.

Multi-Version Concurrency Control (MVCC). MVCC[22–25] ensures that the reader never blocks the writer which leads to better performance. MVCC has been applied in many databases such as SAP HANA[26], Hekaton[27], Deuteronomy[28], Silo[7], BOHM[5], and HyPer[29]. Larson *et al.*[30] revisited the performance of MVCC. These studies can further optimize DOCC's snapshot design and implementation.

Concurrency Control with Hardware Support. There is a trend of applying hardware features to develop high-performance concurrency control. Eris[31] uses programmable switches for sequencing and provides a high-performance in-network concurrency control. DBX[32] uses hardware transactional memory (HTM) to protect OCC's validation phase and commit phase. DrTM[33,34] uses both HTM and remote direct memory access (RDMA) to provide a high-performance concurrency control based on two-phase locking. DrTM+R[11], Fasst[14] and DrTM+H[15] fully leverage the feature of RDMA to accelerate OCC with distributed transactions. These studies are orthogonal to our work, as DOCC can also use programmable switches for determining the transaction execution order, use HTM for protecting DOCC's validation phase and commit phase, and use RDMA for speeding up the network message passing between DOCC's machines.

Fast Concurrency Control for Multicore and Cluster. H-store[12,35] and VoltDB[36] execute transactions serially on one physical unit, which can be a CPU core or a machine in distributed databases. Well-partitioned transactions can achieve good performance. Granola[37] uses a novel distributed coordination protocol to determine transactions' execution order.

Transaction Chopping. Previous work[38–40] shows

the possibility to decompose transactions with analysis of conflict graphs. Lynx [41] and ROCOCO [42] analyze the transactions' chopping graph to reduce latency and to improve parallelism for distributed transactions. DrTM [33, 34] uses transaction chopping to decompose a long-running transaction into smaller pieces, and thus avoids frequent aborts. Callas [43] and IC3 [44] leverage transaction chopping to fully exploit the constrained parallel execution between pieces of transactions. Piece-wise-visibility [45] uses transactions' control-flow graph to exploit the possibility of reading uncommitted data. We believe that DOCC can leverage transaction chopping to exploit transactions' parallelism further.

6 Futrue Work

Although the deterministic database has high scalability, its per-node sequencer which predetermines the transaction order can become a scalability issue when the number of nodes grows very high. At present, a single-thread sequencer is enough for DOCC running on an 8-node cluster. We think this problem can be solved with engineering techniques like multi-threading and treat it as our future work.

Meanwhile, DOCC currently only supports one-shot transactions; therefore DOCC does not work well for workloads that are hard to be partitioned perfectly. Our future work is to generalize DOCC and support general transaction models with minimum overhead.

7 Conclusions

We presented a deterministic concurrency control protocal called deterministic and optimistic concurrency control (DOCC). The first advantage of DOCC is to allow transactions' optimistic execution; thus the determinism is enforced lazily. The second point is that, with snapshot, read-only transaction never blocks the execution, which further improves the performance. Therefore, DOCC is able to improve deterministic database's scalability within one machine. To demonstrate DOCC's effects, we implemented DOCC. The evaluation on TPC-C with a cluster of eight 24-core machines shows that DOCC can outperform state-of-the-art deterministic database, Calvin.

References

[1] Thomson A, Diamond T, Weng S C, Ren K, Shao P, Abadi D J. Calvin: Fast distributed transactions for partitioned database systems. In *Proc. the 2012 ACM SIGMOD International Conference on Management of Data*, May 2012, pp.1-12.

[2] Thomson A, Abadi D J. The case for determinism in database systems. *Proceedings of the VLDB Endowment*, 2010, 3(1): 70-80.

[3] Faleiro J M, Thomson A, Abadi D J. Lazy evaluation of transactions in database systems. In *Proc. the 2014 ACM SIGMOD International Conference on Management of Data*, June 2014, pp.15-26.

[4] Ren K, Thomson A, Abadi D J. Lightweight locking for main memory database systems. *Proceedings of the VLDB Endowment*, 2012, 6(2): 145-156.

[5] Faleiro J M, Abadi D J. Rethinking serializable multiversion concurrency control. *Proceedings of the VLDB Endowment*, 2015, 8(11): 1190-1201.

[6] Fraser K. Practical lock-freedom. Technical Report, University of Cambridge, 2004. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-579.pdf, June 2019.

[7] Tu S, Zheng W, Kohler E, Liskov B, Madden S. Speedy transactions in multicore in-memory databases. In *Proc. the 24th ACM SIGOPS Symposium on Operating Systems Principles*, November 2013, pp.18-32.

[8] Hart T E, McKenney P E, Brown A D, Walpole J. Performance of memory reclamation for lockless synchronization. *Journal of Parallel and Distributed Computing*, 2007, 67(12): 1270-1285.

[9] Arcangeli A, Cao M, McKenney P E, Sarma D. Using read-copy-update techniques for system V IPC in the Linux 2.5 kernel. In *Proc. the 2003 USENIX Annual Technical Conference*, June 2003, pp.297-309.

[10] McKenney P E, Slingwine J D. Read-copy update: Using execution history to solve concurrency problems. In *Proc. the 15th ISCA International Conference on Parallel and Distributed Computing and Systems*, September 2002, pp.509-518.

[11] Chen Y, Wei X, Shi J, Chen R, Chen H. Fast and general distributed transactions using RDMA and HTM. In *Proc. the 11th European Conference on Computer Systems*, April 2016, Article No. 26.

[12] Stonebraker M, Madden S, Abadi D J, Harizopoulos S, Hachem N, Helland P. The end of an architectural era: (It's time for a complete rewrite). In *Proc. the 33rd International Conference on Very Large Data Bases*, September 2007, pp.1150-1160.

[13] Kimura H. FOEDUS: OLTP engine for a thousand cores and NVRAM. In *Proc. the 2015 ACM SIGMOD International Conference on Management of Data*, May 2015, pp.691-706.

[14] Kalia A, Kaminsky M, Andersen D. G. FaSST: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In *Proc. the 12th USENIX Symposium on Operating Systems Design and Implementation*, November 2016, pp.185-201.

[15] Wei X, Dong Z, Chen R, Chen H. Deconstructing RDMA-enabled distributed transactions: Hybrid is better! In *Proc. the 13th USENIX Symposium on Operating Systems Design and Implementation*, October 2018, pp.233-251.

[16] Kung H T, Robinson J T. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 1981, 6(2): 213-226.

[17] Adya A, Gruber R, Liskov B, Maheshwari U. Efficient optimistic concurrency control using loosely synchronized clocks. *ACM SIGMOD Record*, 1995, 24(2): 23-34.

[18] Liskov B, Castro M, Shrira L, Adya A. Providing persistent objects in distributed systems. In *Proc. the 13th European Conference on Object-Oriented Programming*, June 1999, pp.230-257.

[19] Yuan Y, Wang K, Lee R, Ding X, Xing J, Blanas S, Zhang X. BCC: Reducing false aborts in optimistic concurrency control with low cost for in-memory databases. *Proceedings of the VLDB Endowment*, 2016, 9(6): 504-515.

[20] Wang T, Kimura H. Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores. *Proceedings of the VLDB Endowment*, 2016, 10(2): 49-60.

[21] Yu X, Pavlo A, Sánchez D, Devadas S. TicToc: Time traveling optimistic concurrency control. In *Proc. the 2016 ACM SIGMOD International Conference on Management of Data*, June 2016, pp.1629-1642.

[22] Weikum G, Vossen G. Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery (1st edition). Morgan Kaufmann, 2001.

[23] Agrawal D, Sengupta S. Modular synchronization in distributed, multiversion databases: Version control and concurrency control. *IEEE Transactions on Knowledge and Data Engineering*, 1993, 5(1): 126-137.

[24] Bernstein P A, Hadzilacos V, Goodman N. Concurrency Control and Recovery in Database Systems (1st edition). Addison-Wesley, 1987.

[25] Mohan C, Pirahesh H, Lorie R. Efficient and flexible methods for transient versioning of records to avoid locking by read-only transactions. In *Proc. the 1992 ACM SIGMOD International Conference on Management of Data*, June 1992, pp.124-133.

[26] Färber F, Cha S K, Primsch J, Bornhövd C, Sigg S, Lehner W. SAP HANA database: Data management for modern business applications. *ACM SIGMOD Record*, 2012, 40(4): 45-51.

[27] Diaconu C, Freedman C, Ismert E, Larson P Å, Mittal P, Stonecipher R, Verma N, Zwilling M. Hekaton: SQL server's memory-optimized OLTP engine. In *Proc. the 2013 ACM SIGMOD International Conference on Management of Data*, June 2013, pp.1243-1254.

[28] Levandoski J, Lomet D, Sengupta S, Stutsman R, Wang R. High performance transactions in deuteronomy. In *Proc. the 7th Biennial Conference on Innovative Data Systems Research*, January 2015, Article No. 44.

[29] Neumann T, Mühlbauer T, Kemper A. Fast serializable multi-version concurrency control for main-memory database systems. In *Proc. the 2015 ACM SIGMOD International Conference on Management of Data*, May 2015, pp.677-689.

[30] Larson P Å, Blanas S, Diaconu C, Freedman C, Patel J M, Zwilling M. High-performance concurrency control mechanisms for main-memory databases. *Proceedings of the VLDB Endowment*, 2011, 5(4): 298-309.

[31] Li J, Michael E, Ports D R. Eris: Coordination-free consistent transactions using in-network concurrency control. In *Proc. the 26th Symposium on Operating Systems Principles*, October 2017, pp.104-120.

[32] Wang Z, Qian H, Li J, Chen H. Using restricted transactional memory to build a scalable in-memory database. In *Proc. the 9th European Conference on Computer Systems*, April 2014, Article No. 26.

[33] Wei X, Shi J, Chen Y, Chen R, Chen H. Fast in-memory transaction processing using RDMA and HTM. In *Proc. the 25th Symposium on Operating Systems Principles*, October 2015, pp.87-104.

[34] Chen H, Chen R, Wei X, Shi J, Chen Y, Wang Z, Zang B, Guan H. Fast in-memory transaction processing using RDMA and HTM. *ACM Transactions on Computer Systems*, 2017, 35(1): Article No. 3.

[35] Kallman R, Kimura H, Natkins J *et al*. H-store: A high performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment*, 2008, 1(2): 1496-1499.

[36] Stonebraker M, Weisberg A. The voltDB main memory DBMs. *IEEE Data Eng. Bull.*, 2013, 36(2): 21-27.

[37] Cowling J, Liskov B. Granola: Low-overhead distributed transaction coordination. In *Proc. the 2012 USENIX Annual Technical Conference*, June 2012, pp.223-235.

[38] Bernstein P A, Shipman D W. The correctness of concurrency control mechanisms in a system for distributed databases (SDD-1). *ACM Transactions on Database Systems*, 1980, 5(1): 52-68.

[39] Bernstein A J, Gerstl D S, Lewis P M. Concurrency control for step-decomposed transactions. *Information Systems*, 1999, 24(8): 673-698.

[40] Shasha D, Llirbat F, Simon E, Valduriez P. Transaction chopping: Algorithms and performance studies. *ACM Transactions on Database Systems*, 1995, 20(3): 325-363.

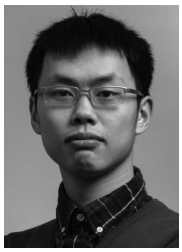
[41] Zhang Y, Power R, Zhou S, Sovran Y, Aguilera M K, Li J. Transaction chains: Achieving serializability with low latency in geo-distributed storage systems. In *Proc. the 24th ACM SIGOPS Symposium on Operating Systems Principles*, November 2013, pp.276-291.

[42] Mu S, Cui Y, Zhang Y, Lloyd W, Li J. Extracting more concurrency from distributed transactions. In *Proc. the 11th USENIX Symposium on Operating Systems Design and Implementation*, October 2014, pp.479-494.

[43] Xie C, Su C, Littley C, Alvisi L, Kapritsos M, Wang Y. High-performance ACID via modular concurrency control. In *Proc. the 25th Symposium on Operating Systems Principles*, October 2015, pp.279-294.

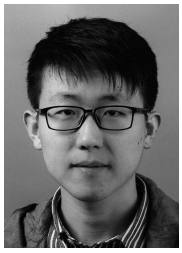
[44] Wang Z, Mu S, Cui Y, Yi H, Chen H, Li J. Scaling multicore databases via constrained parallel execution. In *Proc. the 2016 ACM SIGMOD International Conference on Management of Data*, June 2016, pp.1643-1658.

[45] Faleiro J M, Abadi D J, Hellerstein J M. High performance transactions via early write visibility. *Proceedings of the VLDB Endowment*, 2017, 10(5): 613-624.



Zhi-Yuan Dong received his B.S. degree in software engineering from Shanghai Jiao Tong University, Shanghai, in 2017. He is a Master student and will be a Ph.D. candidate of the Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, Shanghai. His current research interests include distributed system and database systems.

Chu-Zhe Tang was a senior student in software engineering from Shanghai Jiao Tong University, Shanghai. In 2019 fall, he became a Ph.D. candidate of the Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, Shanghai. His research interests include database systems and machine learning.



Jia-Chen Wang received his B.S. degree in software engineering from Nanjing University, Nanjing, in 2018. He is a Master student of the Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, Shanghai. His research interests include database systems and machine learning.

Zhao-Guo Wang received his B.S. degree in software engineering from Nanjing University, Nanjing, in 2008, and his M.S. and Ph.D. degrees in computer science from Fudan University, Shanghai, in 2011 and 2014, respectively. He is an associate professor of the Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, Shanghai. His research interests include distributed systems and storage systems.

Hai-Bo Chen received his B.S. and Ph.D. degrees in computer science from Fudan University, Shanghai, in 2004 and 2009, respectively. He is a professor and the director of the Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, Shanghai. He is a distinguished member of CCF, and a senior member of ACM and IEEE. His research interests include operating systems, and parallel and distributed systems.

Bin-Yu Zang received his B.S., M.S. and Ph.D. degrees in computer science from Fudan University, Shanghai, in 1987, 1990 and 1999 respectively. He is the dean of School of Software, Shanghai Jiao Tong University, Shanghai. He is also a professor of the Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University, Shanghai, and a distinguished member of CCF. His research interests include compilers, computer architecture, operating systems, and parallel and distributed systems.