A Hybrid Approach to Integrating Deterministic and Non-deterministic Concurrency Control in Database Systems

Yinhao Hong
Renmin University of China
Beijing, China
hongyh@ruc.edu.cn

Hongyao Zhao
Renmin University of China
Beijing, China
hongyaozhao@ruc.edu.cn

Wei Lu
Renmin University of China
Beijing, China
lu-wei@ruc.edu.cn

Xiaoyong Du
Renmin University of China
Beijing, China
duyong@ruc.edu.cn

Yuxing Chen
Tencent Inc.
Shenzhen, China
axingguchen@tencent.com

Anqun Pan
Tencent Inc.
Shenzhen, China
aaronpan@tencent.com

Lixiong Zheng
Tencent Inc.
Shenzhen, China
paterzheng@tencent.com

ABSTRACT

Deterministic and non-deterministic concurrency control algorithms have shown respective advantages under diverse workloads. Thus, a natural idea is to blend them together. However, because deterministic algorithms work with stringent assumptions, e.g., batched execution and non-interactive transactions, they hardly work together with non-deterministic algorithms. To address this issue, we propose HDCC, a hybrid approach that adaptively employs Calvin and OCC, which have distinct concurrency control and logging schemes, in the same database system. To ensure serializability and recovery correctness, we introduce lock-sharing, global validation, and two-log-interleaving mechanisms. Additionally, we introduce a rule-based assignment mechanism to dynamically select Calvin or OCC based on workload characteristics. Experimental results using TPC-C and YCSB benchmarks demonstrate that HDCC surpasses existing hybrid approaches by up to 3.1×.

PVLDB Reference Format:
Yinhao Hong, Hongyao Zhao, Wei Lu, Xiaoyong Du, Yuxing Chen, Anqun Pan, and Lixiong Zheng. A Hybrid Approach to Integrating Deterministic and Non-deterministic Concurrency Control in Database Systems.
PVLDB, 18(5): 1376 - 1389, 2025.
doi:10.14778/3718057.3718066

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/dbiir/HDCC.

1 INTRODUCTION

Concurrency control algorithms play a pivotal role in ensuring consistency and isolation in database systems. These algorithms fall into two categories: deterministic and non-deterministic. Deterministic algorithms process transactions in batches, where transactions within each batch are ordered, often based on criteria such as arrival time. A transaction, denoted as T, is scheduled for execution only if all conflict transactions that are ordered prior to T have been completed. Notably, if multiple such transactions exist, they can be executed in parallel, thereby improving concurrency. In contrast, non-deterministic algorithms schedule each transaction individually. The order of concurrent transactions in an equivalent serializable schedule [1] is predetermined at the beginning of a transaction or dynamically determined during execution.

There is no single concurrency control algorithm that is universally optimal for all workloads [2, 18, 51, 55]. Deterministic algorithms [11, 12, 43] excel in processing either high contention workloads or distributed transactions or both, due to their deterministic schedule and without a need for a two-phase commit protocol (2PC). However, deterministic algorithms work under stringent assumptions, like batched execution and predeclared read/write sets, limiting their applicability in reality. Although a few works explore eliminating some of the assumptions, extra overheads are introduced, making them be sub-optimal. For example, Calvin [43] introduces a reconnaissance mechanism by pre-executing the batched transactions, but its performance suffers, especially in the workload with long-running transactions [37]. Aria [29] proposes a deterministic optimistic concurrency control, but at a cost of introducing a 2PC-like protocol as well as a higher abort rate under high contention workload [23, 48]. Worse still, deterministic algorithms cannot support interactive transactions. On the contrary, non-deterministic algorithms such as optimistic concurrency control (OCC) and its variants ([27, 45]), two-phase locking (2PL) and its variants ([3, 17]) work without the aforementioned assumptions. They excel in low or medium contention workloads. However, their performance suffers when processing high contention workloads or distributed transactions [18, 51].

To maximize the respective advantages, we are motivated to develop a hybrid approach that integrates deterministic and non-deterministic algorithms into the same database system. Thus far, although a wide array of hybrid approaches exist, they primarily focus on a combination of non-deterministic algorithms only. Many notable works aim to merge multiple concurrency control algorithms into a singular, innovative algorithm. Examples include MVOCC [8, 26], MV2PL [6, 49] and MVTO [35, 36] that integrate multi-version concurrency control (MVCC) with OCC, 2PL, and TO, respectively. In contrast, other studies [38–41, 46, 50] seek to incorporate multiple concurrency control algorithms within a single database system, with each operating independently and capitalizing on its particular strengths. These algorithms work either

Proceedings of the VLDB Endowment, Vol. 18, No. 5 ISSN 2150-8097.
doi:10.14778/3718057.3718066

at the data item or transaction granularity level. At the data item granularity [41, 42], certain data items, such as frequently accessed (hot) items, are scheduled by one algorithm, while others are handled by different algorithms. At the transaction granularity [38–40, 46], each transaction is scheduled exclusively by a single algorithm. The key design challenge is to ensure serializability among conflict transactions that are scheduled by different concurrency control algorithms. To the best of our knowledge, Snapper [28] is the only work that mixes deterministic and non-deterministic algorithms. In Snapper, transactions with pre-declared read/write sets are scheduled and executed in batches by Calvin, while other transactions are individually scheduled by 2PL. Snapper introduces an extra validation phase for each transaction T scheduled by 2PL, to check whether T can commit or not by comparing it with a batch of transactions scheduled by Calvin. Snapper is sub-optimal because, its concurrency control algorithms do not fully leverage their individual superiority (e.g., Calvin should be used for high contention workloads), and it leads to a higher abort rate due to the coarse-grained validation that takes a batch of transactions as a whole.

In this paper, we propose HDCC, a hybrid approach that incorporates Calvin and OCC into the same database system. We choose Calvin as the deterministic algorithm because of its popularity and adoption in real database systems, like FaunaDB [14]. The reason why we choose OCC as the non-deterministic algorithm is three-fold. First, it is widely used in many real database systems, like Azure Cosmos DB [31, 32, 54]. Second, it excels in low and medium contention workloads [51] and complements Calvin perfectly. Third, if a transaction scheduled by OCC aborts, it can be rescheduled by Calvin due to its known read/write set. In HDCC, each transaction is exclusively scheduled by either Calvin or OCC during its entire execution. The assignment of Calvin and OCC is rule-based, tailoring its optimal choice to different workloads. For example, Calvin is used to process distributed transactions with pre-declared read/write sets, as well as local transactions with pre-declared read/write sets over hot data items.

Because Calvin and OCC adopt completely different concurrency control and logging schemes, integrating them within the same system is not straightforward. The key challenge is to ensure both serializability and correct failure recovery. **(1) Serializability.** Although transactions scheduled by Calvin or OCC individually are serializable, conflict transactions, with some scheduled by Calvin and others by OCC, may not be serializable. Consider transactions T_1 and T_2, scheduled by Calvin and OCC, respectively, involving operations $W_1(x)W_1(y)$ and $R_2(x)R_2(y)$. Assuming that node 1 hosts x and node 2 hosts y, both transactions are decomposed into two sub-transactions each. Conflict arises between T_1 and T_2 in both node 1 and node 2, creating a scenario where T_1 may be ordered before T_2 on node 1 but after on node 2. This conflicting schedule lacks serializability. To solve this problem, we propose two mechanisms, namely lock-sharing and global validation, which are integrated into OCC. Given a transaction T scheduled by OCC, the lock-sharing mechanism in each node helps check which conflict transactions scheduled by Calvin are ordered before T. If T is a distributed transaction, the global validation mechanism further verifies whether T and its conflicting transactions scheduled by Calvin form a cycle across nodes. If a cycle exists, T aborts; otherwise, T is able to commit. Furthermore, by integrating HDCC into the B$^+$-tree index to manage conflicts arising from insert and delete operations alongside range queries, HDCC successfully circumvents phantom anomalies. We theoretically prove that transactions scheduled by HDCC are serializable. Compared to Snapper, HDCC achieves a finer-grained schedule, which leads to a lower abort rate. **(2) Recovery.** Logging is the key technology to ensure failure recovery. However, simply applying the logging mechanisms of Calvin and OCC separately cannot correctly restore the database upon failure. Note that Calvin processes transactions in batches and writes all the logs of batched transactions *before* scheduling them to execute. In contrast, OCC processes each transaction individually and writes the logs *after* scheduling it to execute but before scheduling it to commit. Consequently, the serializable order among transactions scheduled by Calvin and OCC, respectively, cannot be captured based on the logs. When a failure occurs, the committed OCC and Calvin transactions may not be correctly recovered by replaying the redo logs. Intuitively, a baseline approach is to use an additional log file to maintain and record the order once a transaction commits. However, this approach could incur extra maintenance and logging overhead. To address this problem, we design a two-log-interleaving mechanism with a new checkpoint creation process derived from a variant of Zigzag originally introduced in Calvin. In this mechanism, upon the commit of an OCC transaction, we explicitly maintain and log the committed Calvin transactions that are ordered before it. By doing this, we not only capture the order between Calvin and OCC transactions but also record this order in a lightweight manner. Upon a failure, we follow the order recorded in the OCC log and interleave the replay of OCC and Calvin logs to recover the committed Calvin and OCC transactions.

Based on the framework provided by Deneva [18], we integrate HDCC along with state-of-art concurrency control algorithms, including Aria and Snapper, for fair evaluation. Additionally, we enhance Deneva's B$^+$-tree implementation to mitigate phantom anomalies. Leveraging Deneva as the foundation, we conduct a thorough experiment to showcase the performance advantages of HDCC.

To summarize, we make the following contributions.

- We propose HDCC, a hybrid concurrent control algorithm that adaptively employs OCC and Calvin. HDCC is equipped with two mechanisms, lock-sharing and global validation, to ensure the serializable schedule of transactions.
- We design a two-log-interleaving mechanism that helps correctly recover the committed Calvin and OCC transactions once a failure occurs. We provide the theoretical proof to demonstrate its correctness.
- We propose a rule-based assignment mechanism, tailoring its optimal choice to different workloads. Two optimizations are further carefully designed to boost HDCC.
- We conduct extensive experiments over TPC-C and YCSB benchmarks. The results demonstrate that HDCC achieves significantly higher performance compared to the state-of-the-art hybrid approach, with a factor of up to 3.1× improvement.

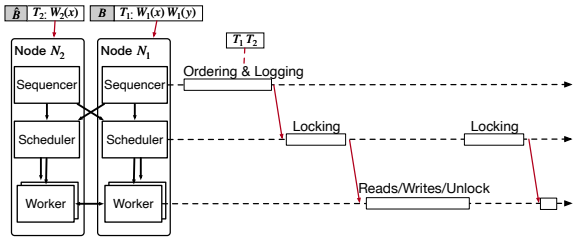


Figure 1: Distributed transaction processing using Calvin

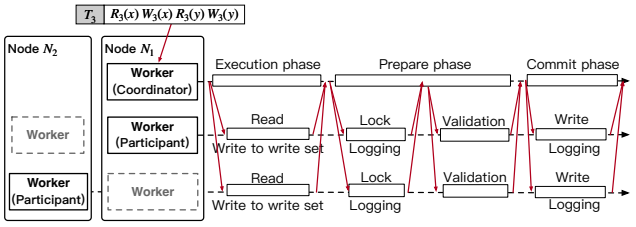


Figure 2: Distributed transaction processing using OCC

2 PRELIMINARY

In this section, we review the concurrency control and logging schemes in Calvin and OCC which are the foundation of HDCC.

2.1 Calvin

In Calvin, each node consists of the Sequencer, Scheduler, and Worker components. The Sequencer collects and packs client transactions into batches, determines their order, and stores them as logical logs on disk in case of failures. For example, consider two transactions, T_1 and T_2, shown in Figure 1. Transaction T_1 writes data item x located in node N_1 and data item y in node N_2, while transaction T_2 writes x. The Sequencer on N_1 (resp. N_2) receives T_1 (resp. T_2), packing it into batches B (resp. $\hat{B}$). Each transaction is allocated a globally unique, monotonically increasing transaction ID. The Sequencers then flush the batched transactions as logical logs on disk and decompose each transaction into sub-transactions based on the nodes involved, sending these sub-transactions to the respective Schedulers. Upon receiving sub-transactions within the same batch, the Scheduler requests locks for them in ascending order of transaction IDs. For instance, the Scheduler on N_1 requests locks for transactions in batches B and $\hat{B}$. After completing the reads/writes, the Worker releases the locks and sends a commit response to its originating Sequencer.

Calvin utilizes a variant of Zigzag [5] to periodically create checkpoints. Data items are flushed to disk only during the checkpoint. Initially, Calvin selects a transaction ID *ckpID* as a pivot. Upon committing all transactions with IDs less than or equal to *ckpID*, the database reaches consistency, and Calvin flushes data items to disk. Transactions with IDs greater than *ckpID* maintain their data as separate copies until the checkpoint is complete. Upon the checkpoint completion, Calvin overwritten data items by the copies. In case of node failure, transactions with IDs greater than *ckpID* are replayed from logical logs for recovery, while those with IDs less than or equal to *ckpID* are already persisted.

2.2 OCC

We adopt Silo [45], a variant of OCC, and extend it to a distributed version. In Silo, a transaction T goes through three phases: execution, prepare, and commit. In the execution phase, Silo performs reads/writes without acquiring locks, and puts the data items that it reads/writes in the read/write set, respectively. In the prepare phase, Silo acquires the locks for its write set and validates whether the data items in its read set have been modified. If no data items have been modified, it enters the commit phase and applies the changes to the database. Note, read-only transactions skip the commit phase.

The coordinator coordinates the execution of a distributed transaction T. T is decomposed into multiple sub-transactions, which are executed locally in the involved nodes using Silo. Consequently, T goes through three phases as well. Take transaction T_3 shown in Figure 2 for example. T_3 is decomposed into two sub-transactions, which are executed on N_1 and N_2, respectively. In the execution phase, the coordinator coordinates N_1 (often in terms of a Worker thread) as one participant in which T_1 reads and writes x without acquiring the lock, and puts x in the read set and write set. Besides, the coordinator coordinates N_2 as the other participant in which T_3 reads/writes y in the same way. After collecting all responses from the participants, the coordinator coordinates T_3 to enter the prepare phase. In the prepare phase, the coordinator notifies N_1 and N_2 to validate. To be specific, the coordinator first enters the locking round and requires N_1 and N_2 to lock all the data items in the write set, i.e., x for N_1 and y for N_2. After acquiring the locks, N_1 and N_2 flush the logs to disk locally. Afterward, the coordinator enters the validation round and sends the validation messages to N_1 and N_2 to check whether the data items in the read set, i.e., x for N_1 and y for N_2, have been modified. Because x and y have never been modified, the coordinator passes the validation. In the commit phase, the coordinator flushes the commit log and notifies N_1 and N_2 to do local commit. N_1 and N_2 write commit logs to disk, release the lock on x and y, and apply the changes to the database.

3 OVERVIEW

HDCC works in a shared-nothing system architecture. Each node is equipped with an in-memory database instance, and logically disaggregated into two layers: the storage layer and the compute layer. An overview of HDCC is given in Figure 3.

Storage layer. The storage layer in our design uses a hash partitioning method [18] to horizontally split data into partitions. Each partition is assigned exclusively to a single node within the cluster, and the storage layer of each node is accountable for managing access to its assigned partitions and writing logs on disk. To support concurrency control, we maintain additional metadata in memory, including lock tables, and statistics on the conflict of data items. This information aids in determining the most suitable concurrency control algorithms for a given transaction.

Compute layer. The compute layer within each node is responsible for selecting either OCC or Calvin to execute transactions, making its optimal choice adaptable to various workload scenarios. For this purpose, it employs four key components:

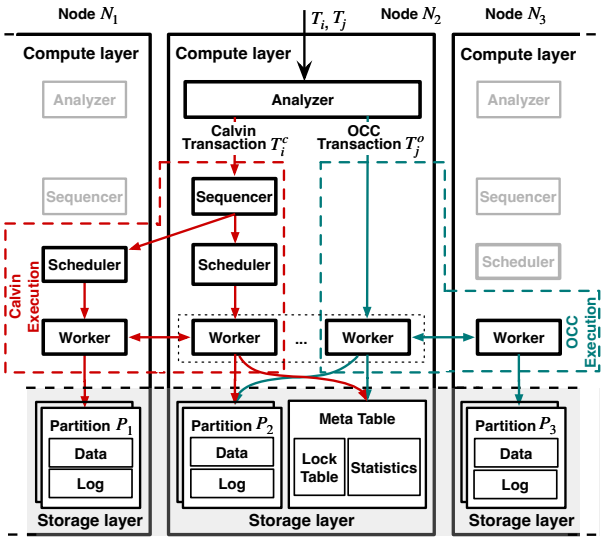


Figure 3: An overview of HDCC

(1) Analyzer. The Analyzer is responsible for receiving and analyzing transactions from clients or users. It analyzes the characteristics of these transactions and assigns the most appropriate algorithms. In our design, this assignment is rule-based, tailoring its optimal choice to different workloads. Transactions scheduled by OCC, referred to as OCC transactions, are directly routed by the Analyzer to Workers that implement OCC for concurrency control. Similarly, transactions scheduled by Calvin, known as Calvin transactions, are directly forwarded by the Analyzer to Sequencer which employs Calvin for concurrency control. For brevity, a transaction T is represented as T^c (resp. T^o) if T is scheduled by Calvin (resp. OCC). Note, each transaction has a unique *ID*, which is a pair $\langle BID, TID \rangle$. *TID* is a monotonically increasing transaction ID assigned by the Analyzer. *BID* signifies the batch ID of the transaction. For Calvin transactions, *BID* will be assigned by the Sequencer later. For OCC transactions, *BID* is set to *NULL*. Given any two $T_i.ID$ and $T_j.ID$, we consider $T_i.ID > T_j.ID$, if $T_i.ID.TID > T_j.ID.TID$.

(2) Sequencer. The Sequencer is responsible for gathering Calvin transactions sent by Analyzers, determining their orders, batching them, and assigning *BID* into their IDs. Then, it writes the logs of these transactions following the same logic as described in Section 2.1. Subsequently, the Sequencer decomposes each transaction in the batch into sub-transactions (if any), and distributes them to the corresponding local or remote schedulers.

(3) Scheduler. The Scheduler is responsible for coordinating the execution of transactions, described in Section 2.1. After acquiring all necessary locks for a sub-transaction T, T is assigned to a single worker for execution. Notably, if there are multiple sub-transactions similar to T that have acquired their locks, they can be executed in parallel. Because conflict transactions in each Scheduler are executed one by one, and they follow the same ascending order of transaction IDs across Schedulers, the schedule of Calvin transactions is serializable.

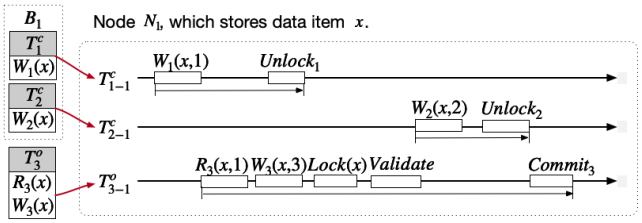


Figure 4: A running example of incorrect local schedule

(4) Multiple workers. A Worker can act as an executor that executes sub-transactions or a coordinator that coordinates the execution of distributed OCC transactions. When a Worker acts as an executor and receives a Calvin sub-transaction from a Scheduler, it follows the same logic as described in Section 2.1 to perform local read and write operations, commits the sub-transaction, and subsequently releases the locks. If the Worker receives an OCC transaction from the Analyzer, it acts as a coordinator and determines whether the transaction is distributed. If the transaction is, it decomposes the transaction into multiple sub-transactions, which are then disseminated to both local and remote Workers for processing; otherwise, it processes the transaction locally. If the Worker receives an OCC sub-transaction from a coordinator, it adopts the role of a participant in distributed transaction processing, as described in Section 2.2. In our design, OCC does not flush dirty pages to disk after commit but relies on checkpoints to persist data as Calvin does. Note, we do not abort Calvin transactions. Instead, we abort OCC transactions as long as we cannot find an equivalent serializable schedule by committing them.

4 HYBRID CONCURRENCY CONTROL

To start, we provide some necessary symbols used throughout the remainder of the paper. Given a transaction T_i^o (resp. T_i^c), let T_{i-k}^o (resp. T_{i-k}^c) be the sub-transaction located in node N_k. Note, in the real execution, for any OCC transaction T_i^o, operations on data items located in N_k could be progressively sent to N_k as the sub-transaction T_{i-k}^o. Given two transactions T_i, T_j, if T_i first, and T_j next operate the same data item x, and one of the two operations is write, we say T_j depends on T_i. This dependency is denoted as $T_i \rightarrow T_j$. *For illustration purposes, when the context is clear, symbols* T_{i-k}^c *and* T_i^c *(resp.* T_{i-k}^o *and* T_i^o*) are used interchangeably.*

4.1 The lock-sharing mechanism

In a single node, there are multiple workers concurrently receiving and executing different sub-transactions. Due to the different execution mechanisms of Calvin and OCC, the conflict cannot directly be detected by each other, leading to inconsistent conflict orders between two sub-transactions.

Example 1. *In Figure 4. There are three sub-transactions in* N_1. T_{1-1}^c: $W_1(x,1)$, T_{2-1}^c: $W_2(x,2)$, *and* T_3^o: $R_3(x)W_3(x)$. *First,* T_{1-1}^c *writes 1 to x, followed by a read (*$x = 1$*) of* T_{3-1}^o. *Second,* T_{3-1}^o *writes x (*$x = 3$*) and adds x to its write set. Third, it enters the prepare phase where* T_{3-1}^o *acquires a lock on x, passes the validation since x has never been modified since its first read, and decides to commit. Fourth, sub-transaction* T_{2-1}^c *updates x to 2, and commits. This*

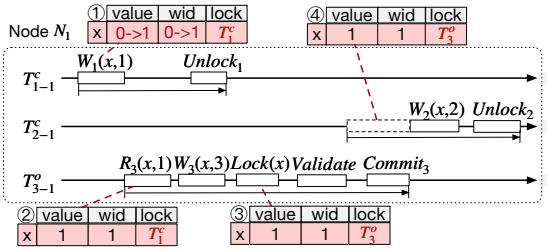


Figure 5: A running example of lock-sharing mechanism

could happen because T^c_{2-1} cannot detect the conflict on x with T^o_{3-1}. Finally, T^o_{3-1} commits and applies its writes to the database. Consequently, the above schedule is not serializable because of the lost update of T^c_{2-1}. □

To address this issue, we introduce the lock-sharing mechanism to unify the metadata between Calvin and OCC so that conflicts among Calvin and OCC transactions can be detected. Specifically, we introduce two elements *lock* and *wid* for each data item in the lock table, operated by both Calvin and OCC. Element *lock* can be either an exclusive lock (EX) or a shared lock (SH). In HDCC, if a Calvin sub-transaction cannot acquire the lock on x, it waits until the lock is released. For an OCC sub-transaction, it requests EX locks in the prepare phase. If any of the locks is held by other sub-transactions, it aborts the whole transaction. And it releases *lock* only after commits. $x.wid$ maintains the $ID.TID$ of the most recent sub-transaction that modifies x. A Calvin sub-transaction updates $x.wid$ to its transaction $ID.TID$ after it writes x. An OCC sub-transaction updates $x.wid$ during its commit phase. In the execution phase, it stores $x.wid$ into its read set when it reads x. In the prepare phase, it checks whether $x.wid$ has been modified by any other sub-transactions after its first read operation on x. If $x.wid$ has been modified, it aborts; otherwise, it checks whether the lock on x is held by any Calvin sub-transaction. If it is, the sub-transaction aborts the whole transaction. For detailed implementation information, please refer to our technical report [10].

Example 2. *Figure 5 illustrates how the lock-sharing mechanism addresses the issue raised in Example 1. T^c_{1-1} first writes x and updates $x.wid$ to 1, which is $T^c_{1-1}.ID.TID$ (step ①). Then, T^o_{3-1} reads x with the value written by T^c_{1-1} (step ②), writes the new value 3 of x in its write set. Next, T^c_{1-1} commits and releases the lock on x. Subsequently, T^o_{3-1} enters the prepare phase, in which it acquires the lock on x, and passes the validation (step ③) because x has not been modified by any other transaction since its first read on x. During the validation, T^c_{2-1} requests the lock on x, and waits for T^o_{3-1} to release the lock (step ④). After T^o_{3-1} commits, T^c_{2-1} acquires the lock on x and writes 2 to x. The above schedule is serializable and the equivalent serializable order is $T^c_{1-1} \rightarrow T^o_{3-1} \rightarrow T^c_{2-1}$.* □

Since a separate schedule of transactions by either Calvin or OCC satisfies conflict serializability, we now prove that a schedule of any two concurrent sub-transactions on the same node using HDCC, with one Calvin transaction, and the other OCC transaction, also ensures conflict serializability. Let $C(T_i)$ be the point in time at which transaction T_i commits. A transaction T_i is said to commit before another transaction T_j if $C(T_i) < C(T_j)$. In HDCC,

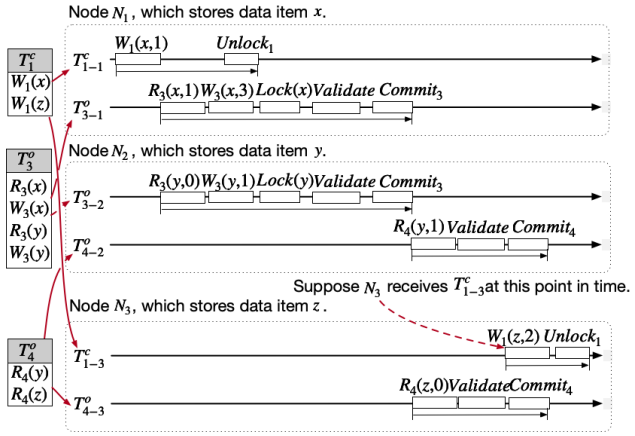


Figure 6: A running example of incorrect global schedule

if T_{i-k} is a Calvin sub-transaction, then the commit time $C(T_{i-k})$ of T_{i-k} is set to the point in time at which T_{i-k} starts to release its locks. If T_{i-k} is an OCC sub-transaction, $C(T_{i-k})$ is set to the point in time at which T_{i-k} enters the commit phase. *Thus, each sub-transaction has one and only one commit time.*

Theorem 1. *In HDCC, given two concurrent sub-transactions T^c_{i-k} and T^o_{j-k} executed on the same node, if $T^c_{i-k} \rightarrow T^o_{j-k}$ (resp. $T^o_{j-k} \rightarrow T^c_{i-k}$), $C(T^c_{i-k}) < C(T^o_{j-k})$ (resp. $C(T^o_{j-k}) < C(T^c_{i-k})$).* □

Theorem 2. *In HDCC, given two concurrent sub-transactions T^c_{i-k} and T^o_{j-k} executed on the same node, the schedule of these two sub-transactions satisfies conflict serializability.* □

Proof sketch. To prove Theorem 1, we list all the six cases where dependencies exist between T^c_{i-k} and T^o_{j-k} in technical report [10]. In each case, we demonstrate that the commitment order aligns with the dependency order of transactions. Given that each sub-transaction only has a singular commit time, Theorem 2 holds true. The formal proof can be found in [10]. □

4.2 The global validation mechanism

Given a set of transactions, the lock-sharing mechanism in HDCC guarantees a serializable schedule of their sub-transactions on each node. However, as shown in Example 3, it cannot ensure that the serializable order of transactions is consistent across nodes.

Example 3. *In Figure 6, T^c_1 updates x on N_1, and z on N_3; T^o_3 updates x on N_1 and y on N_2; T^o_4 reads y on N_2 and z on N_3. On N_1, T^c_{1-1} updates x before T^o_{3-1} does, forming the dependency $T_1 \rightarrow T_3$. On N_2, T^o_{3-2} updates y before T^o_{4-2} reads it, forming the dependency $T_3 \rightarrow T_4$. Additionally, on N_3, T^o_{4-3} reads z before T^c_{1-3} updates it, forming the dependency $T_4 \rightarrow T_1$. Consequently, the dependencies among T_1, T_3, and T_4 form a cycle: $T_1 \rightarrow T_3 \rightarrow T_4 \rightarrow T_1$, indicating that the schedule of these transactions is not conflict-serializable.* □

To prevent forming dependency cycles between Calvin and OCC transactions, we propose the global validation mechanism. This mechanism attempts to obtain $T^o_j.DS^c$, defined in Definition 1, for

each OCC transaction T_j^o, and do the global validation over transactions of $T_j^o.DS^c$ upon the commit of T_j^o. If there exists a transaction $T_i^c \in T_j^o.DS^c$ that has not yet committed, meaning that the committed order between T_i^c and T_j^o is not consistent with the dependency order between them, according to Theorem 1, T_j^o needs to abort; otherwise, T_j^o is able to commit.

Definition 1 (Dependent Set). *Given an OCC transaction T_j^o, the dependent set of T_j^o, denoted as $T_j^o.DS$, is defined as the set of transactions that T_j^o depends on. Among them, the dependent OCC transactions set $T_j^o.DS^o$ is defined as $\{T_i^o | T_i^o \in T_j^o.DS\}$, and the dependent Calvin transactions set $T_j^o.DS^c$ is defined as $\{T_i^c | T_i^c \in T_j^o.DS\}$.* □

Definition 2 (Direct Dependent Set). *The direct dependent set of T_j^o, denoted as $T_j^o.DDS$, is defined as the transactions of $T_j^o.DS$, each T of which takes conflicting access to the same data items with T_j^o, and $T \rightarrow T_j^o$. Among them, the dependent OCC transactions set $T_j^o.DDS^o$ is defined as $\{T_i^o | T_i^o \in T_j^o.DDS\}$, and the dependent Calvin transactions set $T_j^o.DDS^c$ is defined as $\{T_i^c | T_i^c \in T_j^o.DDS\}$.* □

Definition 3 (Indirect Dependent Set). *The indirect dependent set of T_j^o, denoted as $T_j^o.IDS$, is defined as the difference between $T_j^o.DS$ and $T_j^o.DDS$, i.e., $T_j^o.DS - T_j^o.DDS$. Among them, the OCC transaction set $T_j^o.IDS^o$ is $\{T_i^o | T_i^o \in T_j^o.IDS\}$, and the Calvin transaction set $T_j^o.IDS^c$ is $\{T_i^c | T_i^c \in T_j^o.IDS\}$. According to transitive closure, we can have: $T_j^o.IDS = \bigcup_{T_i \in T_j^o.DDS}\{T_i.DS\}$.* □

According to Definition 2 and Definition 3, $T_j^o.DS^c$ is the union of set $T_j^o.DDS^c$ and set $T_j^o.IDS^c$. Acquiring $T_j^o.IDS^c$ requires computing the transitive closure of the indirect dependent set of each transaction in $T_j^o.DDS^c$. When the set $T_j^o.DDS^c$ is relatively large, this computation becomes prohibitively expensive.

Definition 4 (Maximal Calvin Transaction ID). *Given a transaction set S, the maximal Calvin transaction ID of S, denoted as $S.MaxCid$, is defined as the maximal ID of Calvin transactions in S, i.e., $S.MaxCid = \max\{T_i^c.ID | T_i^c \in S\}$.* □

Definition 5 (Maximal Dependent Transaction ID). *The maximal dependent transaction ID of an OCC transaction T_j^o, denoted as $T_j^o.gMaxCid$, is defined as the maximal Calvin transaction ID of $T_j^o.DS^c$, which is $T_j^o.DS^c.MaxCid$.* □

Definition 6 (Extended Dependent Set). *The extended dependent set of T_j^o, denoted as $T_j^o.EDS^c$, is defined as:*
$\{T_i^c | T_i^c.ID \leq T_j^o.gMaxCid\}$. □

As previously mentioned, computing $T_j^o.DS^c$ for T_j^o is expensive. Instead, we attempt to compute a superset of $T_j^o.DS^c$, which is relatively lightweight. Upon the commit of T_j^o, if all transactions in $T_j^o.DS^c$ have committed, then T_j^o is ready to commit; otherwise, T_j^o aborts. To this end, we propose the concept of the maximal dependent transaction ID $T_j^o.gMaxCid$ for T_j^o, as defined in Definition 5, and use $T_j^o.gMaxCid$ to define the superset $T_j^o.EDS^c$ of $T_j^o.DS^c$ in Definition 6. Recall that, given two Calvin transactions T_k^c and T_l^c with $T_k^c.ID < T_l^c.ID$, T_k^c always acquires the locks on the co-accessed data items earlier than T_l^c, and T_l^c commits after T_k^c. Since $T_j^o.EDS^c$ includes the Calvin transaction T with the largest ID in $T_j^o.DS^c$, and all other Calvin transactions with IDs less than or equal to $T.ID$, obviously, $T_j^o.EDS^c$ is a superset of $T_j^o.DS^c$.

Theorem 3. *$\forall T_j^o$, $T_j^o.EDS^c$ is a superset of $T_j^o.DS^c$.* □

Formula 1 shows how to compute $T_j^o.gMaxCid$. We omit the discussion of Equation 1–2, which are self-explanatory. We now discuss how to transform Equation 3 to 4. Recall in Calvin, for any transaction T_m^c, transactions in $T_m^c.DS^c$ must be scheduled to execute before T_m^c. Thus, we have $T_m^c.DS^c.MaxCid < T_m^c.ID$. Since $T_m^c \in T_j^o.DDS^c$, we have $T_m^c.ID < T_j^o.DDS^c.MaxCid$. Therefore, $\{T_m^c.DS^c.MaxCid \mid T_m^c \in T_j^o.DDS^c\}$ in Equation 3 can be omitted.

Formula 1 (Equivalence transformation formula).

$$T_j^o.gMaxCid = T_j^o.DS^c.MaxCid \quad (1)$$

$$= \max(T_j^o.DDS^c.MaxCid, \{T_m.DS^c.MaxCid | T_m \in T_j^o.DDS\} \quad (2)$$

$$= \max(T_j^o.DDS^c.MaxCid, \{T_m^c.DS^c.MaxCid | T_m^c \in T_j^o.DDS^c\}, \{T_n^o.DS^c.MaxCid | T_n^o \in T_j^o.DDS^o\}) \quad (3)$$

$$= \max(T_j^o.DDS^c.MaxCid, \{T_n^o.gMaxCid | T_n^o \in T_j^o.DDS^o\}) \quad (4)$$

For the implementation, we add two elements *cid* and *cid′* associated with each data item x in the lock table. $x.cid$ and $x.cid'$ are used to compute $T_j^o.DDS^c.MaxCid$ and $\max(\{T_n^o.gMaxCid | T_n^o \in T_j^o.DDS^o\})$ in Equation 4 of Formula 1, respectively. Upon any read/write of a Calvin transaction T_i^c, T_i^c would update $x.cid$ properly if $T_i^c.ID > x.cid$. Thus, $x.cid$ always maintains the current maximal Calvin transaction ID on x. In the locking round of the prepare phase in any OCC transaction T_k^o, T_k^o collects the *cid* and *cid′* of each data item that it has ever accessed, and calculates its *gMaxCid* by computing the maximum value of these *cid* and *cid′*. Upon the commit of T_k^o, for each data item x that it has ever accessed, T_k^o sets $x.cid'$ to $\max\{x.cid', T_k^o.gMaxCid\}$. In the validation round of the prepare phase, T_j^o conducts global validation, which checks whether all the Calvin sub-transactions with IDs smaller or equal to $T_j^o.gMaxCid$ have been committed at each node accessed by T_j^o. If successful, T_j^o commits; otherwise, it aborts. Due to space limitations, we leave the discussion of the implementations in our technical report [10].

Example 4. *Continuing Example 3, Figure 7 shows how the global validation mechanism prevents the dependency cycles. In step ①, when T_{1-1}^c writes x, it sets $x.cid$ to $T_1^c.ID$, which is $\langle 1,1\rangle$. In step ②, T_3^o conducts the prepare phase, in which it collects $x.cid$, $x.cid'$, $y.cid$, and $y.cid'$. Next, T_3^o computes $T_3^o.DDS^c.MaxCid$, which is $\langle 1,1\rangle$, and $max\{T_n^o.gMaxCid | T_n^o \in T_3^o.DDS^o\}$, which is $\langle 0,0\rangle$. According to Equation 4, $T_3^o.gMaxCid$ is set to $\langle 1,1\rangle$. In step ③, upon T_3^o commits, it updates $x.cid'$ and $y.cid'$ to $T_3^o.gMaxCid$, which is $\langle 1,1\rangle$. In step ④, following the same procedure as T_3^o, T_4^o calculates its gMaxCid, which is $\langle 1,1\rangle$. Then, T_4^o detects that transaction T_1^c with its ID equal to $T_4^o.gMaxCid$ on N_3 has not committed, and then T_4^o aborts, breaking the dependency cycle $T_1 \rightarrow T_3 \rightarrow T_4 \rightarrow T_1$.* □

Now we prove that the schedule of transactions using HDCC is conflict serializable in Theorem 4, and Theorem 5.

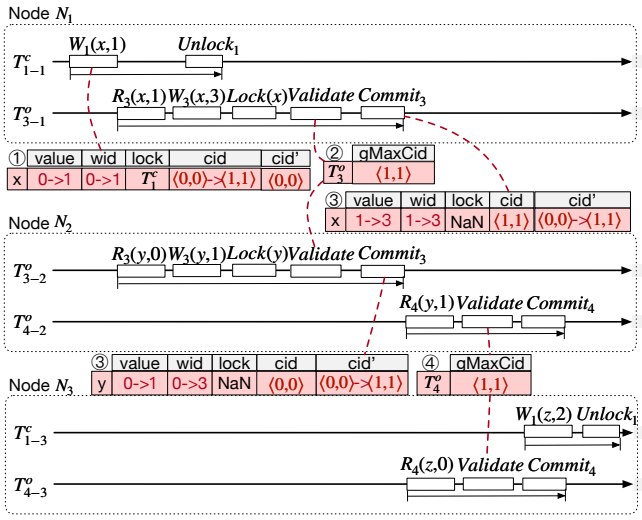


Figure 7: A running example of global validation

Theorem 4. *In HDCC, given any two committed transactions T^c_i and T^o_j, on each node N_k where there exists a dependency between T^c_{i-k} and T^o_{j-k}, T^c_{i-k} and T^o_{j-k} must have a consistent order of either $C(T^c_{i-k}) < C(T^o_{j-k})$ or $C(T^o_{j-k}) < C(T^c_{i-k})$.* □

Theorem 5. *In HDCC, the schedule of any two concurrent transactions T^c_i and T^o_j is conflict serializable.* □

Proof sketch. we prove both theorems by contradiction. If the conclusion of Theorem 4 does not hold, according to Theorem 1, there would exist two distinct dependencies between T^c_i and T^o_j, which conflicts with the global validation mechanism. If the conclusion of Theorem 5 does not hold, the sub-transactions of T^c_i and T^o_j form a cycle, contradicting Theorem 4. The formal proof can be found in [10]. □

4.3 Optimizations

To further enhance HDCC's performance, we propose a rule-based assignment mechanism along with two additional optimizations.

4.3.1 The rule-based assignment. In HDCC, the scheduling of transactions by either Calvin or OCC depends on their specific characteristics. Since Calvin assumes that a transaction must predeclare its read/write set, the first rule dictates that any transaction T without a predeclared read/write set is scheduled using OCC.

Rule 1. *A transaction without pre-declaring its read/write set is scheduled by OCC.* □

As discussed, Calvin excels at handling distributed transactions, while OCC struggles. To address this, we propose Rule 2.

Rule 2. *A distributed transaction that has declared its read/write set is scheduled by Calvin.* □

Calvin excels in handling high-contention workloads, whereas OCC is more effective in low- to medium-contention scenarios. In HDCC, the contention level of a workload is measured by counting the number of conflicts in each partition $\mathcal{P}_x$. A conflict occurs when any transaction in $\mathcal{P}_x$ interferes with others, increasing its conflict count. $\mathcal{P}_x$ is marked as *hot* if its conflict count exceeds a predefined threshold within a specified period. Based on this definition of partition "temperature," we propose two heuristic rules, Rule 3 and Rule 4, to assign concurrency control algorithms.

Rule 3. *A transaction with a declared read/write set is scheduled by Calvin if it accesses hot partition(s).* □

Rule 4. *A transaction that does not satisfy the requirements of Rule 1–3 is scheduled by OCC.* □

4.3.2 Re-schedule of aborted OCC transactions. An OCC transaction typically aborts during its validation phase. At this point, we can collect its complete read/write set, which meets the requirements for Calvin. This allows the analyzer to reassign the scheduling algorithm as described in Section 4.3.1. Note that after restarting an aborted OCC transaction and rescheduling it with Calvin, the read/write set may change due to concurrent transactions, potentially causing another abort. However, as pointed out in [43], such issues are more common when transactions rely on secondary indexes for read/write set access. Since secondary indexes are rarely used in frequently updated fields, continuous transaction aborts are uncommon.

4.3.3 Immediate read and deferred commit. Consider an OCC transaction T^o_j reads data item x written by a Calvin transaction T^c_i. This case can lead to two schedules. The first schedule is that T^c_i commits before T^o_j. In this case, T^o_j can commit as discussed in Section 4.1. In the second schedule, T^o_j enters the prepare phase before T^c_i commits. In the original design, T^o_j would detect that x is locked by T^c_i and aborts. However, in HDCC, Calvin transactions do not abort due to conflicts. Thus, we defer the decision instead of immediate abort to reduce false positive aborts. In the prepare phase, T^o_j is deferred if all these conditions are met: (1) at least one data item x in the read set is locked by a T^c_i; (2) $x.wid$ remains unmodified after T^o_j's first read of x; and (3) $x.wid$ equals to $T^c_i.ID$. If deferred, T^o_j records the ID of T^c_i and waits for T^c_i commits. If T^c_i aborts, T^o_j cascading aborts; otherwise, T^o_j commits. Note, although HDCC does not introduce additional aborts for Calvin transactions, according to [43], a Calvin transaction may abort in two cases: (1) If there is a mismatch between the actual and planned read/write sets, and (2) If the transaction logic explicitly requires an abort after reading.

5 FAILURE RECOVERY

In this section, we first give the general idea, then present the two-log-interleaving mechanism, and finally discuss its correctness.

5.1 General idea

Upon any system failure, we need to perform redo and undo operations to ensure data consistency. The redo operation follows the serializable orders of committed transactions to recover the writes of them. The undo operation rolls back the modification of transactions that had not yet committed before the failure.

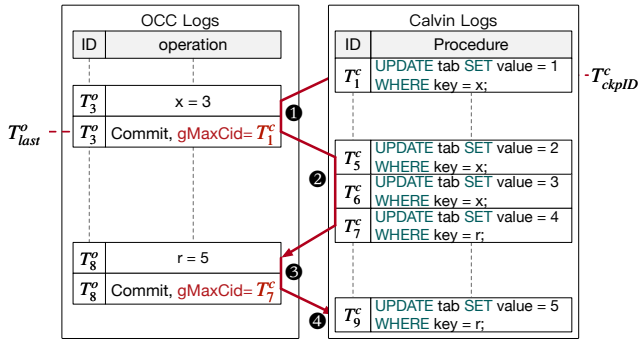


Figure 8: Two-log-interleaving mechanism

In HDCC, no undo operations are needed because only data from committed transactions is flushed to disk; uncommitted transactions' data is never flushed. In our design, HDCC periodically creates checkpoints. During each checkpointing, for Calvin transactions, we select a transaction T^c_{ckpID} and flush the data items written by transactions with IDs less than or equal to $T^c_{ckpID}.ID$. Note that Calvin does not write dirty pages, but flushes data items to disk. We do not flush writes from transactions with IDs greater than $T^c_{ckpID}.ID$. Instead, we maintain these writes as copies. Details of checkpointing Calvin transactions are discussed in Section 2.1. When all the Calvin transactions with IDs less than or equal to $T^c_{ckpID}.ID$ commit, the database reaches a consistency point. For OCC transactions, those committed before this point are included in the checkpoint, while those committed afterward must store their data in separate copies. The last committed OCC transaction included in the checkpoint is recorded as T^o_{last} for failure recovery. Note that during checkpointing, an OCC transaction with its *gMaxCid* smaller than $T^c_{ckpID}.ID$ can execute normally, while one with its *gMaxCid* larger than $T^c_{ckpID}.ID$ must wait to commit until the database reaches the consistency point. This wait is necessary because these OCC transactions rely on a Calvin transaction not included in the checkpoint. Since the commit time of these OCC transactions closely aligns with the consistency point, this delay is not expected to impact overall throughput, as supported by the experiment in Section 7.2.3.

In HDCC, we need to perform redo operations. Unlike traditional databases that have only a single category of logs, HDCC maintains both OCC logs and Calvin logs. Because OCC logs and Calvin logs are separate, the commit order between OCC transactions and Calvin transactions is lost. Therefore, to perform redo operations, we first need to recover the commit order between OCC transactions and Calvin transactions, and then replay the logs according to the ascending commit order of transactions.

5.2 Two-log-interleaving mechanism

HDCC has two types of logs, Calvin logs and OCC logs, as shown in Figure 8. Calvin logs record transactions' logical logs, including their IDs and procedures. For example, transaction T^c_1 records its procedure like "UPDATE tab SET value = 1 WHERE key = x". OCC logs contain redo logs and commit logs for OCC transactions. Redo logs record the new value of modified data items from committed OCC transactions, while commit logs confirm that OCC transactions have successfully committed. We maintain $T^o.gMaxCid$ in the commit log of T^o. Since $T^o.gMaxCid$ represents the maximum ID of dependent Calvin transactions that commit before T^o, by maintaining *gMaxCid*, we capture the commit order between OCC and Calvin transactions. As shown in Figure 8, T^o_3 records its redo log as $\langle T^o_3 : x = 3\rangle$, and its commit log as $\langle T^o_3 : Commit, gMaxCid = T^c_1\rangle$, indicating that T^o_3 needs to be recovered after T^c_1.

On any node N, upon a system failure, after N restarts, our two-log-interleaving mechanism follows an interleaved manner to replay the logs of committed transactions. (1) We sequentially scan the OCC logs to identify the first OCC transaction T^o_j ordered after T^o_{last} that needs to be recovered. (2) We sequentially scan Calvin logs and replay the logs of all Calvin transactions with their IDs smaller than or equal to $T^o_j.gMaxCid$, but greater than $T_{ckpID}.ID$. (3) We replay the OCC logs of transaction T^o_j, and find the next OCC transaction T^o_{j+1} in OCC logs. (4) We continue to scan Calvin logs and replay each Calvin transaction T^c_i where $T^c_i.ID$ falls within the range $T^o_j.gMaxCid < T^c_i.ID \leq T^o_{j+1}.gMaxCid$. (5) We repeat step 3 to 4 until all OCC transactions involved in the OCC logs have been recovered. (6) We restore the other Calvin transactions by replaying the rest of Calvin logs.

Example 5. *As depicted in Figure 8, the solid red lines indicate the recovery sequence of logs in HDCC upon a failure. Due to a checkpoint created beforehand, HDCC can recover from the two transactions T^c_{ckpID} (T^c_1), and T^o_{last} (T^o_3). HDCC first sequentially scans the OCC logs to find the first OCC transaction T^o_8 that needs to be recovered (step ❶). Since the $T^o_8.gMaxCid$ is T^c_7, meaning that HDCC needs to recover T^c_7 before recovering T^o_8. Thus, HDCC sequentially scans Calvin logs and replays the Calvin transactions from T^c_5 to T^c_7 (step ❷). Subsequently, HDCC recovers the transaction T^o_8. Upon completion, HDCC finds all OCC transactions involved in OCC logs have been recovered (step ❸). At this stage, HDCC restores the other Calvin transaction T^c_9 by replaying its Calvin log (step ❹).*

Note that in the original Calvin logging mechanism, the logs recorded by nodes are incomplete because they only capture transactions received, not those executed, meaning if a node fails, it may lack the logs for transactions it needs to execute. To address this, HDCC modifies Calvin's logging mechanism. Upon receiving a sub-transaction, the node will persist that sub-transaction, ensuring that each node maintains a complete set of sub-transactions for execution. Furthermore, during the redo of a local Calvin transaction T^c_i, its writes may rely on the data read from other nodes that might have changed due to subsequent transactions. Thus, the remote nodes are required to replay transactions to retrieve the previous data, disrupting execution on those nodes. To tackle this, HDCC introduces Calvin partial redo logs. When a Calvin transaction's writes depend on a remote read, the outcome of that write is logged in the Calvin redo logs. Remote nodes keep their read data until the write is saved to handle potential failures. Note that maintaining all the writes of Calvin transactions in the redo logs is insufficient for failure recovery since Calvin requires the recovery of all transactions within a batch to ensure determinism. Uncommitted transactions cannot record their redo logs when failure occurs,

making failure recovery impossible only using redo logs. Therefore, these uncommitted transactions must rely on logical logs for failure recovery. Since Calvin requires logical logs for failure recovery, we do not recommend maintaining a global redo log, as it would further increase log overhead.

5.3 Correctness

We prove the correctness of our failure recovery in Theorem 6 and provide its TLA+ specification [25] in our technical report [10].

Theorem 6. *In HDCC, the transaction schedule for failure recovery is conflict equivalent to that of the original execution.* □

Proof sketch. Since *gMaxCid* shows the order between Calvin and OCC transactions, the recorded order of conflicting transactions in the log matches their original execution order. For transactions without conflicts, their log order is always equivalent to their original order. Thus, all transaction orders in the log are conflict equivalent to the original execution order. Since the two-log-interleaving mechanism replays transactions based on log order, the theorem holds. □

6 IMPLEMENTATION

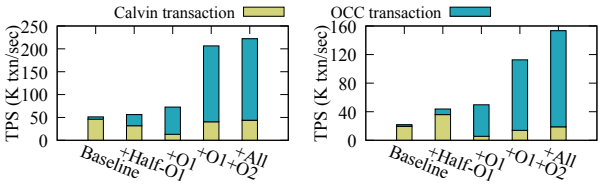
This section explains how we build HDCC. We implement HDCC based on Deneva[18]. As a unified testbed for distributed concurrency control algorithms, Deneva already has algorithms including OCC and Calvin. It horizontally splits data across nodes and supports both hash/B$^+$-tree indexes. We improve Deneva by: 1) Enhancing the B$^+$-tree indexes to avoid phantom by applying concurrency control to the leaf nodes, following [45, 56]. 2) Introducing a new thread to monitor the contention level of workloads. 3) Implementing Aria and Snapper into Deneva for fair comparisons.

We implement HDCC by integrating OCC and Calvin. Since the lock-sharing mechanism unifies the metadata of OCC and Calvin, and the global validation mechanism is designed to integrate into the prepare phase of OCC. The implementation of HDCC follows two steps: 1) defining unified metadata and creating an interface for interacting with it, including locking, unlocking, changing data versions, and validation; 2) building the specific logic for HDCC using these interfaces, including two critical tasks: locking Calvin transactions within the Scheduler and handling the prepare phase of OCC. Note that we choose single-transaction granularity for logging in HDCC, as batch granularity is primarily designed to enhance thread scalability on a single machine, which is not the focus of HDCC.

We employ B$^+$-tree indexes to support range queries in HDCC. To address phantom reads, transactions use the lock-sharing mechanism to lock the leaf nodes of the B$^+$-tree during insertions or deletions. During range queries, transactions adhere to the lock-sharing mechanism by locking or validating the leaf nodes in the range. If conflicts arise, the transaction may be blocked or fail validation, thus avoiding phantom reads.

7 EVALUATION

In this section, we conduct a comprehensive experimental evaluation of HDCC and the following research questions are addressed:



(a) Effect of Optimization (YCSB) (b) Effect of Optimization (TPC-C)

Figure 9: Optimization analysis

(1) How effective are the optimizations, the performance under dynamic workloads, and the scalability, including the logging overhead and checkpoint creation overhead, in HDCC? (Section 7.2)

(2) How does HDCC perform compared with its constituent algorithms (i.e., Calvin and OCC) and the state-of-the-art algorithms (e.g., Aria and Snapper)? (Section 7.3)

7.1 Experimental setup

Test platform. We deploy a share-nothing in-memory database system with 2 servers, each equipped with an 18-core (36-thread) 2.20 GHz Intel Xeon Gold 5220 processor and 156 GB memory. Each server runs on 64-bit CentOS Stream 9 with Linux kernel 5.14.0 and GCC 11.5.0. They are located in the same rack and connected via a 1000 Mbps switch, with an average network latency of 0.12 ms. Each server installs a client and a database. The database uses 16 worker threads, 1 sequencer thread, 1 scheduler thread, 1 analyzer thread, and other threads for inter-node communication and transaction management. The client has four worker threads for generating transactions and other threads for communication. For the scalability experiments in Section 7.2.3, we use Tencent Cloud with 12 S5.6XLARGE48 servers. Each node has 24 vCPUs and 48 GB memory, with latencies between servers ranging from 0.1 to 0.5 ms.

Workloads. Our experiments are conducted with two benchmarks, namely YCSB [7] and TPC-C [44]. For YCSB, we generate about 8 million records for each database node and each transaction executes 10 read/write operations that access data items following the Zipfian distribution, with a default skew factor of 0.9 and a write-read rate of 0.2 (i.e., 20% writes and 80% reads). For TPC-C, unless stated otherwise, we generate 32 warehouses for each database node and only use NewOrder and Payment transactions. The other transaction types are single-node transactions that are not suitable for our evaluation and represent a smaller proportion. When certain transaction types of TPC-C are excluded, the ratio between the remaining transactions increases proportionally.

Configurations. In addition, we introduced two metrics: the distributed transaction rate, and the undeclared transaction rate. The distributed transaction rate quantifies the proportion of transactions accessing data across multiple nodes, while the undeclared transaction rate measures the proportion of transactions that are unaware of their read/write sets. By default, we set the distributed transaction rate to 20% for the YCSB workload. For the TPC-C workload, we use the default values, setting the distributed transaction rate to 10% for NewOrder transactions and 15% for Payment transactions. Unless otherwise stated, we set the undeclared transaction rate to 0.

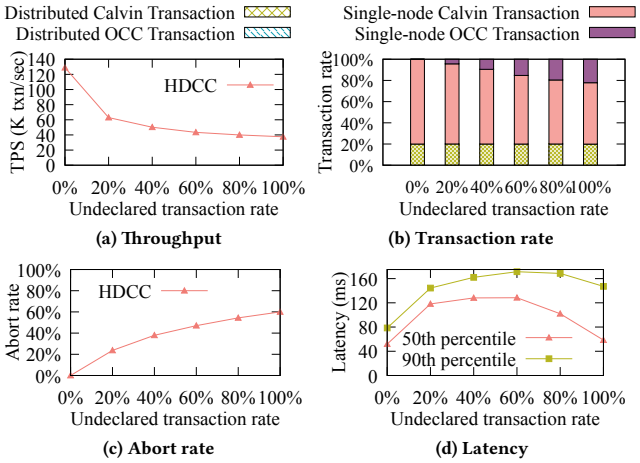


Figure 10: The analysis of *O2*

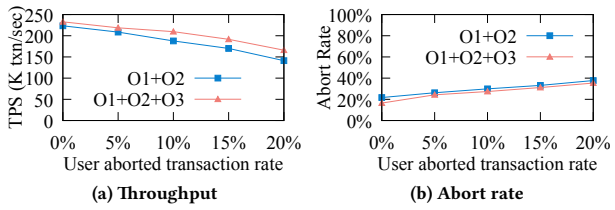


Figure 11: The analysis of *O3*

7.2 The effeciency of HDCC

7.2.1 Effect of optimizations. We assess the effectiveness of the proposed optimizations described in Section 4.3: ***Half-O1*** (rule 1, 3), ***O1*** (rules 1–4), ***O2*** (re-scheduling of aborted OCC transactions), and ***O3*** (immediate read and deferred commit). By default, we set the undeclared transaction rate to 10% for both YCSB and TPC-C.

Figure 9a shows the throughput (TPS) of various optimizations on the YCSB. *Half-O1*, *O1*, *O1+O2*, and *All* (*O1+O2+O3*) improve the throughput by 0.1×, 0.4×, 3.0×, and 3.3×, respectively, compared to the basic assignment utilized in Snapper (labeled ***Baseline***). The enhancement from *Half-O1* stems from directing transactions accessing hot partitions to Calvin, suitable for high-conflict scenarios, while assigning other transactions to OCC, suitable for low-conflict scenarios. *O1* boosts performance by assigning all distributed transactions to Calvin, leveraging its strengths. The effectiveness of *O2* helps maintain read/write sets of aborted OCC transactions, allowing HDCC to potentially assign these transactions to Calvin for optimization. *O3* allows OCC transactions to read uncommitted data from Calvin, reducing abort rates. Figure 9b shows similar trends for TPC-C, consistent with the results from YCSB.

We evaluate the *O2* optimization, which contributes the most to the throughput improvement, to assess its overhead. Figure 10 shows results in a high-conflict scenario (skew factor = 1.3) as the undeclared transaction rate varies from 0 to 100%. When the undeclared transaction rate is 0%, the transaction latency is minimal, around 50 ms at the median percentile. As the undeclared transaction rate increases, throughput declines but stabilizes around 40 K transactions per second, with median and 90th percentile latencies rising to 2.2× compared to the 0%. We consider this additional overhead acceptable since *O2* maintains relatively high performance even with a higher undeclared transaction rate. Notably, when the undeclared transaction rate exceeds 60%, both the median and 90th percentile latencies decrease. Figure 10b shows that many single-node OCC transactions are committed in these cases, leading to lower overall latency.

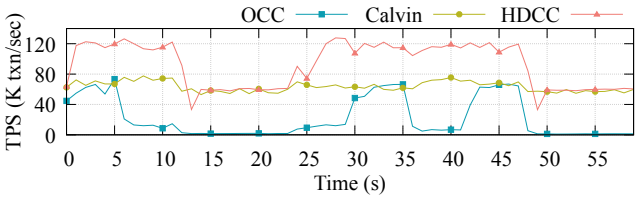


Figure 12: Effect of dynamic workload (YCSB)

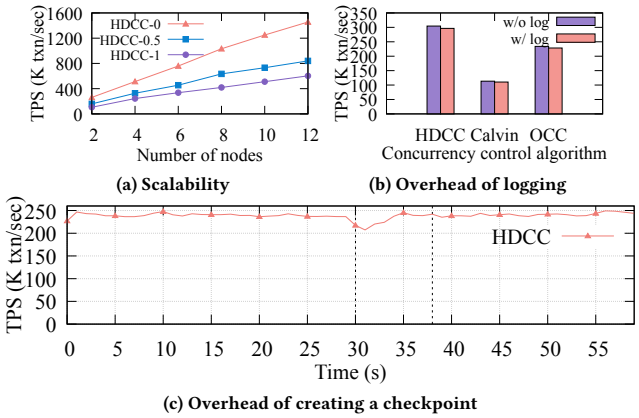
As described in Section 4.3.3, OCC transactions may face cascading aborts with *O3*. We specifically evaluated scenarios where Calvin transactions might abort, as shown in Figure 11. By varying the user aborted transaction rate from 0% to 20%, we found a 37% decrease in throughput for *O1+O2*. In contrast, *O1+O2+O3* exhibited a 29% decrease. This suggests that the throughput gains from *O3* outweigh the overhead from cascading aborts. Figure 11b supports this, showing a lower abort rate for *O1 + O2 + O3* compared to *O1 + O2*. For the following evaluations, HDCC adopts all optimizations.

7.2.2 Dynamic workload. To evaluate HDCC's adaptability to online dynamic workloads, we use YCSB for a 60-second experiment under dynamic conditions. We switch the YCSB workload configurations every 6 seconds, randomly selecting a write-read rate from 0.1, 0.5, and 0.9, and a skew factor from 0.3, 0.9, and 1.5. Figure 12 shows the results, where switches occur at timestamps 7, 13, and 25, etc. HDCC responds promptly and dynamically adjusts the algorithm assignment based on the changing workload. After each switch, the throughput of HDCC rapidly aligns to be better than or comparable to OCC or Calvin.

7.2.3 Scalability, overhead of logging, and checkpoint creation. We evaluate the scalability of HDCC using YCSB and TPC-C workloads by varying the number of nodes from 2 to 12. We report results for three undeclared transaction rates, that is: 0 (HDCC-0), 0.5 (HDCC-0.5), and 1 (HDCC-1). We only report YCSB results due to similar findings. Figure 13a shows that HDCC-0, HDCC-0.5, and HDCC-1 exhibit linear scalability. From 2 to 12 nodes, HDCC-0 increased by 5.57X, HDCC-0.5 by 5.3X, and HDCC-1 by 5.72X, demonstrating excellent scalability.

Figure 13b illustrates that the logging mechanisms we implemented for durability incurred minor and comparable overhead. The throughput reduction with logging was negligible for OCC, Calvin, and HDCC, at 2.4%, 2.7%, and 2.7%, respectively. For fairness, we do not compare throughput with logging in Section 7.3 as Aria and Snapper have not specified logging mechanisms.

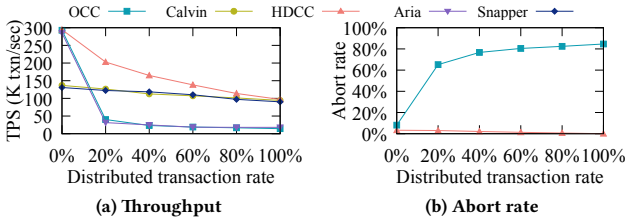
Figure 13c shows the overhead of creating a checkpoint, which occurs between the two black dotted lines from 30 to 38 seconds. The minimum throughput during this period is only about 10%



(a) Scalability (b) Overhead of logging

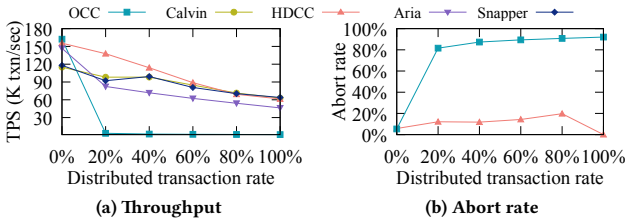
(c) Overhead of creating a checkpoint

Figure 13: Scalability, overhead of logging, and checkpoint creation (YCSB)



(a) Throughput (b) Abort rate

Figure 14: Effect of distributed transaction rate (YCSB)



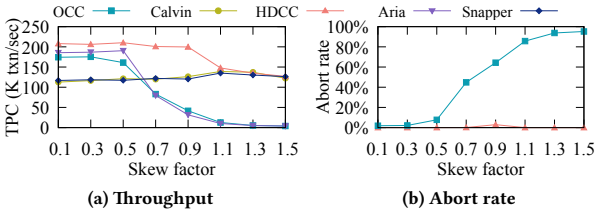
(a) Throughput (b) Abort rate

Figure 15: Effect of distributed transaction rate (TPC-C)

lower than the average, indicating that saving the checkpoint to disk has little impact on throughput. This is due to the asynchronous nature of the checkpoint process. The decrease in throughput is primarily due to resource contention for the CPU and disk.

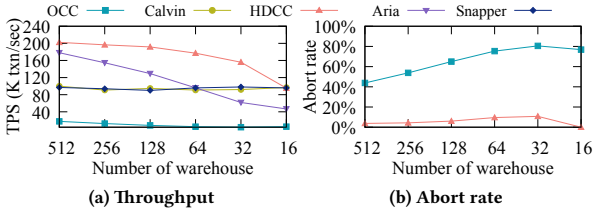
7.3 Compared with mainstream algorithms

7.3.1 Effect of distributed transaction rate. As shown in Figure 14a, when the distributed transaction rate is 0, HDCC, OCC, and Aria exhibit similar high throughput, surpassing Calvin and Snapper. In this case, HDCC, Aria, and OCC can execute single-node transactions without needing the 2PC protocol, , while Calvin and Snapper are limited by a single Scheduler. As the distributed transaction rate increases, Calvin and Snapper maintain stable throughput, while Aria, OCC, and HDCC see a decrease. Calvin and Snapper perform similarly because when the undeclared transaction rate is set to 0, Snapper assigns all transactions to Calvin. HDCC,



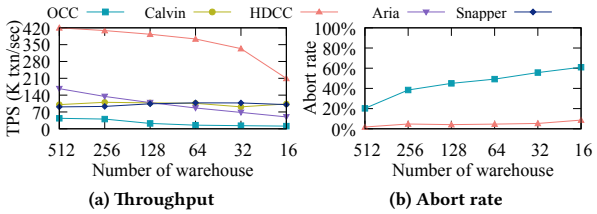
(a) Throughput (b) Abort rate

Figure 16: Effect of the contention level (YCSB)



(a) Throughput (b) Abort rate

Figure 17: Effect of the contention level (TPC-C)



(a) Throughput (b) Abort rate

Figure 18: Effect of the contention level (Full TPC-C)

using Calvin for distributed transactions through *O1*, exhibits a smaller drop in throughput. In contrast, OCC and Aria decline significantly due to the 2PC protocol, which lengthens execution times for distributed transactions. Note, both Calvin and Aria are deterministic concurrency control, and Snapper assigns all transactions to Calvin in this experiment, hence they do not have any abort rates in Figure 14b. When the distributed transaction rate approaches 100%, the throughput of HDCC becomes comparable to that of Calvin, and is about 6.8× higher than that of OCC and Aria. At this stage, most transactions in HDCC are Calvin transactions. The results of the TPC-C benchmark share a similar trend to that of YCSB, as presented in Figure 15.

7.3.2 Effect of the contention level. Figure 16 shows the throughput and abort rate as the skew factor (SF) varies from 0.1 to 1.5. At low contention levels (SF <= 0.5), the performance of all five algorithms remains relatively stable. HDCC outperforms other algorithms due to the optimizations. At moderate contention levels (0.5 < SF < 1.1), OCC and Aria experience a decline in throughput, while HDCC maintains stable throughput mainly due to optimization *O2*. At high contention levels (SF >= 1.1), most transactions in HDCC are scheduled by Calvin, leading to a similar throughput to Calvin. The performance of Snapper remains similar to that of Calvin. We also tested the TPC-C workload by varying the number of warehouses to simulate different contention levels (fewer warehouses indicate higher contention). As shown in Figure 17, the

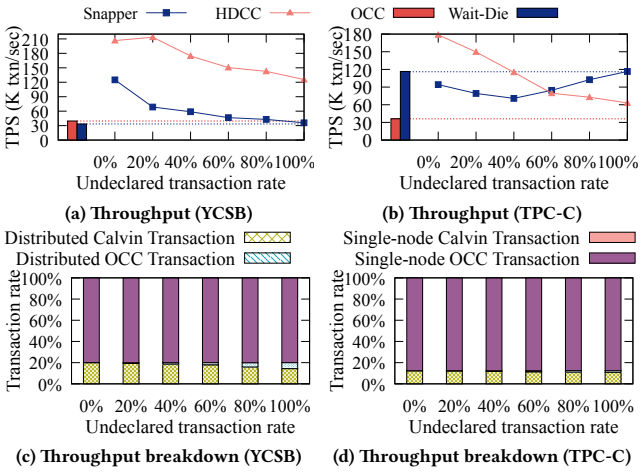


Figure 19: Effect of undeclared transaction rate

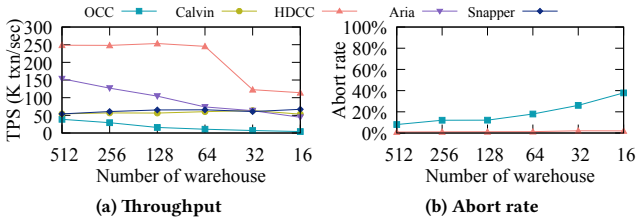


Figure 20: Effect of range queries

throughput trend of HDCC is similar to that of YCSB. Moreover, we experiment with a full TPC-C workload, as shown in Figure 18, the throughput trend is also similar, and the throughput of OCC and HDCC algorithms are higher than when only NewOrder and Payment transactions are used. This increase is due to the lower contention levels in the full TPC-C workload.

7.3.3 Effect of undeclared transaction rate. This part compares the performance of HDCC with Snapper by varying the undeclared transaction rate from 0 to 1. Figure 19a shows that when all transactions pre-declare their read/write sets, HDCC outperforms Snapper, achieving up to 3.1× higher throughput. This is because *O1* assigns OCC to execute all local transactions even when they are feasible for Calvin (refer to Figure 19c for throughput breakdown). In contrast, Snapper executes Calvin as long as all read/write sets are pre-declared. As the undeclared transaction rate increases, the throughput of HDCC slightly declines, but due to the *O2* and *O3* optimization, still superior to Snapper. Figures 19b and 19d show similar results on the TPC-C dataset, with HDCC achieving up to 2.3 times higher throughput. The exceptions are when the undeclared transaction rate exceeds 0.8, Snapper performs better than HDCC. This is primarily due to the Wait-Die algorithm used in Snapper, which performs better than OCC in this scenario [18]. The Wait-Die algorithm's superior conflict handling leads to Snapper's overall better performance in this case.

7.3.4 Effect of range queries. In this section, we assess the impact of range queries and insert operations. We use a workload of 92% NewOrder and 8% StockLevel transactions from TPC-C, with NewOrder involving insert operations and StockLevel including range queries on the same table. Figure 20 shows that the throughput trends under varying contention levels align with those in Section 7.3.2, indicating that range queries does not change the previous results.

8 RELATED WORK

This section studies the related work of concurrency control algorithms and hybrid concurrency control algorithms.

Concurrency control. Concurrency control algorithms can be divided into two categories: (1) deterministic algorithms and (2) non-deterministic algorithms. The first category executes transactions in a scheduled deterministic order. The classical algorithms like Calvin [43] make stringent assumptions that read/write sets of transactions must be pre-declared. Recent works, such as Aria [29], BCDB [33], Harmony [24] and DOCC [9], designs optimistic deterministic concurrency control to execute transactions first and then determine the order. These methods eliminate the stringent assumption but introduce 2PC-like communication. The second category is non-deterministic algorithms, including 2PL [3, 4, 15, 17], OCC [16, 21, 22, 27, 30, 45, 52, 53], timestamp ordering [4], MVCC [13, 34–36, 49]. However, these algorithms inevitably require the 2PC protocol to address the serializability of distributed transactions, introducing significant network communication overhead. Furthermore, these algorithms also suffer from a substantial number of aborts in high contention scenarios [18, 19].

Hybrid concurrency control. Hybrid approaches aim to merge multiple algorithms into a singular one or incorporate multiple algorithms within a single database system. The examples of the former include MOCC [47], MVOCC [26], which integrate 2PL and MVCC with OCC, respectively. The latter mostly focuses on incorporating non-deterministic algorithms. HSync [38] and C3 [40] hybrid 2PL and OCC by dividing transactions into different categories. CormCC [41] mixes PartCC [20], 2PL, and OCC by dividing data partitions into different categories. Tebaldi [39] constructs hierarchical concurrency control by analyzing the stored procedures. Polyjuice [46] introduces reinforcement learning to design specific concurrency control for each stored procedure. Snapper [28] is the only work that mixed deterministic and non-deterministic (i.e., 2PL) algorithms.

9 CONCLUSION

In this paper, we present HDCC, a novel distributed hybrid approach that integrates Calvin and OCC. We propose lock-sharing and global validation mechanisms to guarantee the serializability of HDCC. Then we design a two-log-interleaving mechanism to ensure correct recovery upon failure. Besides, we propose a rule-based assignment mechanism, tailoring its optimal choice to different workload scenarios. Through comprehensive experimentation on YCSB and TPC-C benchmarks, HDCC is proven to excel over the state-of-the-art hybrid approach by providing up to 3.1× better throughput.

REFERENCES

[1] Atul Adya, Barbara Liskov, and Patrick E. O'Neil. 2000. Generalized Isolation Level Definitions. In *Proceedings of the 16th International Conference on Data Engineering, San Diego, California, USA, February 28 - March 3, 2000*, David B. Lomet and Gerhard Weikum (Eds.). IEEE Computer Society, 67–78. https://doi.org/10.1109/ICDE.2000.839388

[2] Rakesh Agrawal, Michael J. Carey, and Miron Livny. 1987. Concurrency Control Performance Modeling: Alternatives and Implications. *ACM Trans. Database Syst.* 12, 4 (1987), 609–654. https://doi.org/10.1145/32204.32220

[3] Claude Barthels, Ingo Müller, Konstantin Taranov, Gustavo Alonso, and Torsten Hoefler. 2019. Strong consistency is not hard to get: Two-Phase Locking and Two-Phase Commit on Thousands of Cores. *Proc. VLDB Endow.* 12, 13 (2019), 2325–2338.

[4] Philip A. Bernstein and Nathan Goodman. 1981. Concurrency Control in Distributed Database Systems. *ACM Comput. Surv.* 13, 2 (1981), 185–221.

[5] Tuan Cao, Marcos Antonio Vaz Salles, Benjamin Sowell, Yao Yue, Alan J. Demers, Johannes Gehrke, and Walker M. White. 2011. Fast checkpoint recovery algorithms for frequently consistent applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, Timos K. Sellis, Renée J. Miller, Anastasios Kementsietsidis, and Yannis Velegrakis (Eds.). ACM, 265–276. https://doi.org/10.1145/1989323.1989352

[6] Yuxing Chen, Anqun Pan, Hailin Lei, Anda Ye, Shuo Han, Yan Tang, Wei Lu, Yunpeng Chai, Feng Zhang, and Xiaoyong Du. 2024. TDSQL: Tencent Distributed Database System. *Proc. VLDB Endow.* 17, 12 (2024), 3869–3882.

[7] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, Joseph M. Hellerstein, Surajit Chaudhuri, and Mendel Rosenblum (Eds.). ACM, 143–154. https://doi.org/10.1145/1807128.1807152

[8] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias (Eds.). ACM, 1243–1254. https://doi.org/10.1145/2463676.2463710

[9] Zhiyuan Dong, Chuzhe Tang, Jia-Chen Wang, Zhaoguo Wang, Haibo Chen, and Binyu Zang. 2020. Optimistic Transaction Processing in Deterministic Database. *J. Comput. Sci. Technol.* 35, 2 (2020), 382–394. https://doi.org/10.1007/s11390-020-9700-5

[10] Hongyin Hao et al. 2024. A hybrid approach to integrating deterministic and non-deterministic concurrency control in database systems (technical report). https://github.com/dbiir/HDCC/blob/master/HDCC.pdf.

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

[12] Jose M. Faleiro and Daniel J. Abadi. 2015. Rethinking serializable multiversion concurrency control. *Proc. VLDB Endow.* 8, 11 (2015), 1190–1201. https://doi.org/10.14778/2809974.2809981

[13] Jose M. Faleiro and Daniel J. Abadi. 2015. Rethinking serializable multiversion concurrency control. *Proc. VLDB Endow.* 8, 11 (2015), 1190–1201.

[14] Fauna. [n.d.]. Fauna | The Distributed Serverless Database. Retrieved March 7, 2025 from https://fauna.com/

[15] Jim Gray, Raymond A. Lorie, Gianfranco R. Putzolu, and Irving L. Traiger. 1976. Granularity of Locks and Degrees of Consistency in a Shared Data Base. In *IFIP Working Conference on Modelling in Data Base Management Systems*. North-Holland, 365–394.

[16] Jinwei Guo, Peng Cai, Jiahao Wang, Weining Qian, and Aoying Zhou. 2019. Adaptive Optimistic Concurrency Control for Heterogeneous Workloads. *Proc. VLDB Endow.* 12, 5 (2019), 584–596.

[17] Zhihan Guo, Kan Wu, Cong Yan, and Xiangyao Yu. 2021. Releasing Locks As Early As You Can: Reducing Contention of Hotspots by Violating Two-Phase Locking. In *SIGMOD Conference*. ACM, 658–670.

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

[19] Yihe Huang, William Qian, Eddie Kohler, Barbara Liskov, and Liuba Shrira. 2020. Opportunities for Optimism in Contended Main-Memory Multicore Transactions. *Proc. VLDB Endow.* 13, 5 (2020), 629–642.

[20] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alex Rasin, Stanley B. 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 (2008), 1496–1499. https://doi.org/10.14778/1454159.1454211

[21] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2016. ERMIA: Fast Memory-Optimized Database System for Heterogeneous Workloads. In *SIGMOD Conference*. ACM, 1675–1687.

[22] H. T. Kung and John T. Robinson. 1981. On Optimistic Methods for Concurrency Control. *ACM Trans. Database Syst.* 6, 2 (1981), 213–226.

[23] Z. Lai, H. Fan, W. Zhou, Z. Ma, X. Peng, F. Li, and E. Lo. 2023. Knock Out 2PC with Practicality Intact: a High-performance and General Distributed Transaction Protocol. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. IEEE Computer Society, Los Alamitos, CA, USA, 2317–2331. https://doi.org/10.1109/ICDE55515.2023.00179

[24] Ziliang Lai, Chris Liu, and Eric Lo. 2023. When Private Blockchain Meets Deterministic Database. *Proc. ACM Manag. Data* 1, 1 (2023), 98:1–98:28. https://doi.org/10.1145/3588952

[25] Leslie Lamport. 1994. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.* 16, 3 (may 1994), 872–923. https://doi.org/10.1145/177492.177726

[26] Per-Åke 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 (2011), 298–309. https://doi.org/10.14778/2095686.2095689

[27] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. 2017. Cicada: Dependably Fast Multi-Core In-Memory Transactions. In *SIGMOD Conference*. ACM, 21–35.

[28] Yijian Liu, Li Su, Vivek Shah, Yongluan Zhou, and Marcos Antonio Vaz Salles. 2022. Hybrid Deterministic and Nondeterministic Execution of Transactions in Actor Systems. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 65–78. https://doi.org/10.1145/3514221.3526172

[29] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2020. Aria: A Fast and Practical Deterministic OLTP Database. *Proc. VLDB Endow.* 13, 11 (2020), 2047–2060. http://www.vldb.org/pvldb/vol13/p2047-lu.pdf

[30] Hatem A. Mahmoud, Vaibhav Arora, Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. 2014. MaaT: Effective and scalable coordination of distributed transactions in the cloud. *Proc. VLDB Endow.* 7, 5 (2014), 329–340. http://www.vldb.org/pvldb/vol7/p329-mahmoud.pdf

[31] Microsoft. [n.d.]. Azure Cosmos DB - NoSQL and Relational Database | Microsoft Azure. Retrieved March 7, 2025 from https://azure.microsoft.com/en-us/products/cosmos-db/

[32] MonetDB. [n.d.]. MonetDB. Retrieved March 7, 2025 from http://www.monetdb.org

[33] Senthil Nathan, Chander Govindarajan, Adarsh Saraf, Manish Sethi, and Praveen Jayachandran. 2019. Blockchain Meets Database: Design and Implementation of a Blockchain Relational Database. *Proc. VLDB Endow.* 12, 11 (2019), 1539–1552. https://doi.org/10.14778/3342263.3342632

[34] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *SIGMOD Conference*. ACM, 677–689.

[35] David P. Reed. 1978. *Naming and synchronization in a decentralized computer system*. Ph.D. Dissertation. Massachusetts Institute of Technology, Cambridge, MA, USA. https://hdl.handle.net/1721.1/16279

[36] David P. Reed. 1983. Implementing Atomic Actions on Decentralized Data. *ACM Trans. Comput. Syst.* 1, 1 (1983), 3–23. https://doi.org/10.1145/357353.357355

[37] 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 (jun 2014), 821–832. https://doi.org/10.14778/2732951.2732955

[38] Zechao Shang, Feifei Li, Jeffrey Xu Yu, Zhiwei Zhang, and Hong Cheng. 2016. Graph Analytics Through Fine-Grained Parallelism. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 463–478. https://doi.org/10.1145/2882903.2915238

[39] Chunzhi Su, Natacha Crooks, Cong Ding, Lorenzo Alvisi, and Chao Xie. 2017. Bringing Modular Concurrency Control to the Next Level. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, Semih Salihoglu, Wenchao Zhou, Rada Chirkova, Jun Yang, and Dan Suciu (Eds.). ACM, 283–297. https://doi.org/10.1145/3035918.3064031

[40] Xuebin Su, Hongzhi Wang, and Yan Zhang. 2021. Concurrency Control Based on Transaction Clustering. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*. IEEE, 2195–2200. https://doi.org/10.1109/ICDE51399.2021.00223

[41] Dixin Tang and Aaron J. Elmore. 2018. Toward Coordination-free and Reconfigurable Mixed Concurrency Control. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, Haryadi S. Gunawi and Benjamin Reed (Eds.). USENIX Association, 809–822. https://www.usenix.org/conference/atc18/presentation/tang

[42] Dixin Tang, Hao Jiang, and Aaron J. Elmore. 2017. Adaptive Concurrency Control: Despite the Looking Glass, One Concurrency Control Does Not Fit All. In *8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2017/papers/p63-tang-cidr17.pdf

[43] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*, K. Selçuk Candan, Yi Chen, Richard T. Snodgrass, Luis Gravano, and Ariel Fuxman (Eds.). ACM, 1–12. https://doi.org/10.1145/2213836.2213838

[44] TPC. [n.d.]. TPC-C Homepage. Retrieved March 7, 2025 from http://www.tpc.org/tpcc/

[45] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013*, Michael Kaminsky and Mike Dahlin (Eds.). ACM, 18–32.

[46] Jia-Chen Wang, Ding Ding, Huan Wang, Conrad Christensen, Zhaoguo Wang, Haibo Chen, and Jinyang Li. 2021. Polyjuice: High-Performance Transactions via Learned Concurrency Control. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, Angela Demke Brown and Jay R. Lorch (Eds.). USENIX Association, 198–216. https://www.usenix.org/conference/osdi21/presentation/wang-jiachen

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

[48] Yang Wang, Miao Yu, Yujie Hui, Fang Zhou, Yuyang Huang, Rui Zhu, Xueyuan Ren, Tianxi Li, and Xiaoyi Lu. 2022. A Study of Database Performance Sensitivity to Experiment Settings. *Proc. VLDB Endow.* 15, 7 (mar 2022), 1439–1452. https://doi.org/10.14778/3523210.3523221

[49] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An Empirical Evaluation of In-Memory Multi-Version Concurrency Control. *Proc. VLDB Endow.* 10, 7 (2017), 781–792. https://doi.org/10.14778/3067421.3067427

[50] Chao Xie, Chunzhi Su, Cody Littley, Lorenzo Alvisi, Manos Kapritsos, and Yang Wang. 2015. High-performance ACID via modular concurrency control. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, Ethan L. Miller and Steven Hand (Eds.). ACM, 279–294. https://doi.org/10.1145/2815400.2815430

[51] 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 (2014), 209–220. https://doi.org/10.14778/2735508.2735511

[52] Xiangyao Yu, Andrew Pavlo, Daniel Sánchez, and Srinivas Devadas. 2016. TicToc: Time Traveling Optimistic Concurrency Control. In *SIGMOD Conference*. ACM, 1629–1642.

[53] Xiangyao Yu, Yu Xia, Andrew Pavlo, Daniel Sánchez, Larry Rudolph, and Srinivas Devadas. 2018. Sundial: Harmonizing Concurrency Control and Caching in a Distributed OLTP Database Management System. *Proc. VLDB Endow.* 11, 10 (2018), 1289–1302.

[54] INC. YUGABYTE. [n.d.]. YugabyteDB. The Distributed SQL Database for Mission-Critical Apps. Retrieved March 7, 2025 from https://www.yugabyte.com/

[55] Hongyao Zhao, Jingyao Li, Wei Lu, Qian Zhang, Wanqing Yang, Jiajia Zhong, Meihui Zhang, Haixiang Li, Xiaoyong Du, and Anqun Pan. 2024. RCBench: an RDMA-enabled transaction framework for analyzing concurrency control algorithms. *The VLDB Journal* 33, 2 (2024), 543–567.

[56] Zhanhao Zhao, Hongyao Zhao, Qiyu Zhuang, Wei Lu, Haixiang Li, Meihui Zhang, Anqun Pan, and Xiaoyong Du. 2023. Efficiently Supporting Multi-Level Serializability in Decentralized Database Systems. *IEEE Transactions on Knowledge and Data Engineering* 35, 12 (2023), 12618–12633. https://doi.org/10.1109/TKDE.2023.3277969