Fine-Grained Re-Execution for Efficient Batched Commit of Distributed Transactions (Extended Version)

Zhiyuan Dong†, Zhaoguo Wang†, Xiaodong Zhang†, Xian Xu†
Changgeng Zhao⋄, Haibo Chen†, Aurojit Panda⋄, Jinyang Li⋄
† Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
⋄ Department of Computer Science, New York University

Abstract

Distributed transaction systems incur extensive cross-node communication to execute and commit serializable OLTP transactions. As a result, their performance greatly suffers. Caching data at nodes that execute transactions can cut down remote reads. Batching transactions for validation and persistence can amortize the communication cost during committing. However, caching and batching can significantly increase the likelihood of conflicts, causing expensive aborts. In this paper, we develop Hackwrench to address the challenge of caching and batching. Instead of aborting conflicted transactions, Hackwrench tries to repair them using *fine-grained re-execution* by tracking the dependencies of operations among a batch of transactions. Tracked dependencies allow Hackwrench to selectively invalidate and re-execute only those operations necessary to "fix" the conflict, which is cheaper than aborting and executing an entire batch of transactions. Evaluations using TPC-C and other micro-benchmarks show that Hackwrench can outperform existing commercial and research systems including FoundationDB, Calvin, COCO, and Sundial under comparable settings.

1 Introduction

Distributed system supporting serializable OLTP transactions is a crucial component of the cloud's storage infrastructure. Over the past decade, many distributed transaction systems have been proposed and deployed, with notable examples including Spanner [16], CockroachDB [15], H-Store [30], and FoundationDB [65]. However, while these systems can scale across many nodes, their achieved performance still leaves much to be desired.

There are fundamental reasons why distributed transactions tend to be slow. As data is partitioned across multiple nodes, the system often must fetch data from a remote node (aka remote reads) during transaction execution. More importantly, in order to commit a transaction, the system must also coordinate across multiple nodes to ensure serializability. Such coordination can show up in the form of distributed 2PL-style locking [16] or OCC-style [4] validation, followed by two-phase commit (2PC) [4, 16, 40]. Consequently, executing and committing a transaction requires multiple round trips of blocking communication. This is disastrous for performance, resulting in significantly reduced throughput, especially for contended workloads.

To substantially boost performance, we aim to drastically cut down the amount of remote communication needed to execute and commit a distributed transaction.Caching and batching are promising techniques in realizing our goal. Caching data extensively at nodes that execute transactions can reduce remote reads. Batching a group of transactions together for validation and commit can amortize the communication needed across multiple transactions. These techniques are already used ubiquitously among single machine databases. Recently, the cloud database Aurora [53, 54] achieves impressive performance using caching and batching. However, single-master Aurora's simple setting of executing transactions on a single database node makes it much easier to apply caching and batching with good performance results.

Caching and batching have seen limited use in a distributed setting where multiple nodes can execute and commit transactions simultaneously. This is because both techniques can significantly increase the likelihood of non-serializable interleaving under contention. Sinfonia [4] performs best-effort caching. However, under contention, cached reads can miss writes recently committed by other nodes, causing the corresponding transactions to abort. When transactions are batched together for validation, the invalidation of a transaction due to conflicts will cause other transactions in the batch to abort. Thus, COCO [40] validates transactions individually and only batches validated transactions for cross-node replication. Addressing these challenges is critical to enabling the effective use of caching and batching, but how to do so has remained an open question.

In this paper, we propose Hackwrench, a distributed transaction system designed for OLTP workloads. Hackwrench performs best-effort caching and batched validation to reduce remote communication while mitigating the harmful effect of increased conflicts. Our key idea is to "repair" non-serializable transactions by applying a minimal fix rather than naively aborting an entire batch of transactions. Transactions are fixed by re-executing operations that have read stale data and are thus invalidated.

To support fine-grained re-execution, Hackwrench transactions are expressed as a dataflow graph of operations to make their dependencies explicit. Hackwrench introduces a tiered commit protocol: transactions first go through a local *commit* phase to resolve conflicts within a database node, and then a global *commit* phase to resolve conflicts among different database nodes. A database node uses a traditional local concurrency control mechanism (e.g., 2PL [26]) to execute and commit transactions locally. Transactions can read uncommitted data of any locally-committed transaction without waiting for its global commit, so that they will not be blocked. The resulting dependencies are tracked by the database node and are used later during repair.

The global commit protocol validates and commits a batch of locally committed transactions. It works similarly to the two-phase commit protocol: in the *prepare* phase, the database node contacts all participating storage nodes to validate the reads and persist the batch's writes as well as transaction inputs at the storage nodes; in the *commit* phase, the database node notifies storage nodes of the batch's commit status. Hackwrench's global commit introduces two variations to this basic protocol. First, upon validation failure, the database node tries to repair the batch using the updated cache; it re-executes affected operations according to the tracked dependencies within and

among transactions, and includes the delta between the original and repaired write set in the commit message. Second, Hackwrench relies on a timestamp server to determine the commit order of transactions. Storage nodes validate transactions in the order of their commit timestamps, which guarantees that repair only needs to happen at most once.

For a common but restricted class of transactions called one-shot transactions [30, 45], we can optimize the global commit by offloading the repair to storage nodes. A one-shot transaction's dataflow graph can be decomposed into several independent pieces. Hackwrench's fast-path optimization leverages this feature to let a storage node immediately commit a batch of transactions upon receiving its prepare request, repairing if necessary, without waiting for the commit message. This optimization allows storage nodes to handle prepare messages without blocking, avoiding two-phase commit costs.

We have implemented Hackwrench as a distributed transaction system and compared its performance with a baseline OCC system, FoundationDB [65], Calvin [49], COCO [40], and Sundial [63] Using a cluster of 19 machines, our TPC-C evaluation shows that Hackwrench's performance gains over existing systems in terms of throughput can be up to 730.03% (OCC), 1889.52% (FDB), 385.57% (COCO), 470.60% (Calvin) and 45.18%(Sundial) when the fraction of multi-warehouse NewOrder exceeds 89% (§ 5). Further performance analysis shows that Hackwrench's fine-grained repair mechanism can greatly reduce the overhead of aborts when commits are batched. To summarize, the paper makes the following contributions:

- We introduce a new system design, Hackwrench, for distributed OLTP transactions. Hackwrench exploits batching and caching to reduce communication during transaction execution and commit. In particular, we propose a tiered commit protocol to validate and commit a batch of transactions in two stages, ensuring serializability first within the local node that has executed the transactions and then across all nodes globally. To mitigate increased conflicts due to stale cache reads and batched validation, we propose fine-grained re-execution to "fix" stale or invalidated reads instead of doing traditional wholesale abort-and-retry. Doing so greatly lowers the cost of transaction conflicts.
- For one-shot transactions [30, 45], we propose the fast-path optimization which performs re-execution at storage instead of database nodes with one fewer round-trip and no 2PC coordination cost.
- We build a prototype of Hackwrench and show that it can outperform existing commercial and research systems including FoundationDB [65], COCO [40], Calvin [49], and Sundial [63] under comparable settings.

2 Background and Motivation

We discuss the cost of distributed OLTP transactions, explain the performance challenges faced by two promising techniques, caching and batching, and motivate our approach.

2.1 The Cost of Distributed OLTP Transactions

Distributed OLTP transactions incur heavy performance costs because of their intrinsic need for cross-machine communication. There are two sources of communication. The first is remote reads, incurred during transaction execution when data are not available locally. For "remote storage" systems, aka systems that execute transactions on machines separate from storage servers (e.g., Spanner [16], FoundationDB [65], CockroachDB [15], and MySql NDB Cluster [43]), all reads are remote. For "co-located" systems, aka systems that execute transactions on worker threads co-located with storage servers (e.g. Calvin [49], H-Store [30], COCO [40], and Sundial [63]), a fraction of the reads in a "multi-partition" transactions[1] must contact some remote server.

The second is remote synchronization, needed for ensuring serializability, that can take several forms: i) distributed two-phase locking [26], e.g., used by Spanner [16]); ii) OCC validation [31], e.g., used by COCO [40]; iii) two-phase commit (2PC) [23, 25], which ensures that a committed transaction's data is durable on all relevant servers. To reduce round-trips, many systems also merge OCC validation with the first phase of 2PC, e.g., Sinfonia [4], Granola [17].

Compared to local execution, remote communication drastically increases transaction latency and decreases system throughput as well. Throughput is particularly affected when a limited number of transactions can be run concurrently to mask the increased transaction latency. This could either be due to the lack of sufficiently many transaction-issuing clients, or due to the workload having inherently limited concurrency. For example, in the TPC-C workload, only a few transactions can execute concurrently without conflicts in each warehouse.

2.2 Challenges of Caching and Batching

We work with a "remote-storage" system architecture in which database nodes that execute transactions are separate from storage nodes that store partitioned data. During transaction execution, database nodes read from storage nodes and buffer writes locally. To commit a transaction, the database node must first validate its reads with the relevant storage nodes. Below, we discuss how the two common performance-optimization techniques, caching and batching, can be applied in this setting to reduce cross-node communication and the challenges in realizing their performance potential.

Caching. Database nodes can keep a local cache of previously accessed data and read cached data to avoid remote reads to storage nodes. In the setting where only a single database can process write transactions (e.g. Deuteronomy [33], single-master Aurora [53]), the cache is always consistent and up-to-date. However, in our setting, different database nodes can commit transactions that write to the same data, resulting in stale/inconsistent cache [2]. For correctness, one can check the cached reads' validity during the commit validation, as done in Sinfonia [4]. Thus, caching increases transaction aborts, which may explain why most systems choose not to cache [15, 16, 43, 65].

Batching. To amortize the cost of remote synchronization, database nodes can batch a group of transactions after they finish execution to validate and commit them together at the storage nodes. Prior works have proposed batching transaction commits, but not validation. For example, single-writer Aurora [53, 54] only batches the writes of committed transactions for replication to storage nodes. COCO [40] first performs OCC validation for individual transactions and then batches a group of validated transactions for replication. In our setting, database nodes must validate a transaction's reads at storage nodes, a process that involves cross-node synchronization. Therefore, to amortize this cost, it is imperative to batch the validation of a group of transactions. Many tough design questions arise. Should we permit a

[1] Multi-partition transactions access multiple data partitions stored on different servers.

[2] Traditional cache consistency protocols [22, 34] are not robust against failures. Thus, it is more practical to adopt a best-effort cache that is updated or invalidated asynchronously in the background with no guaranteed consistency

p	TPC-C Throughput (Txns/s)			
	Naive OCC	+Caching	+Batching	+RU
0%	19.0k	32.3k	103.3k	337.7k
9.6%	18.3k	29.7k	72.6k	442
89.3%	13.7k	21.7k	28.2k	47

Table 1: The performance impact of caching and batching. p is the percentage of multi-warehouse NewOrder transaction. "Naive OCC" is similar to COCO [40], except there is no co-location of transaction execution with storage. "+Caching" adds a local cache at each database node. "+Batching" further makes database nodes batch transaction validation and commit. Finally, "+RU" permits reading uncommitted data between different batches.

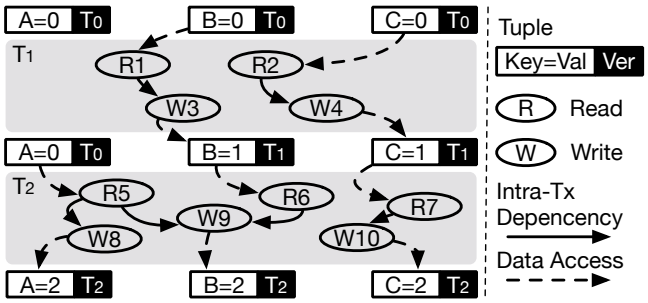


Figure 1: Fine-grained tracking of operation dependencies within a batch of transactions, T_1 and T_2.

transaction to read uncommitted writes from those that are still waiting for batch validation to complete at remote nodes? Does an entire batch need to be aborted if one transaction in the batch fails the validation?

We conducted experiments on the TPC-C benchmark to quantify the performance impact of caching and batching. The experiments use 18 Amazon EC2 m5.2xlarge instances, among which 6 are database nodes and 12 are storage nodes. We increase the likelihood of cross-node conflicts by increasing the multi-warehouse NewOrder transaction possibility (p). The results are shown in Table 1. Caching increases throughput by approximately $58.4\% \sim 69.8\%$. Batching further improves performance by 219.5% ($p = 0\%$) and 144.2% ($p = 9.6\%$). However, when cross-node conflicts are common ($p = 89.3\%$), the improvement drops to 30.3% due to i) batched validation increases the likelihood of conflicts between batches, ii) a single aborted transaction causes the entire batch of transactions to abort. Basic batching does not allow a transaction to read uncommitted data of those batches still in the process of validation with remote storage nodes. We also experimented with a variation (+RU) that allows reading from such uncommitted batches. When there is no contention ($p = 0\%$), this design achieves significantly higher throughput than basic batching ($337.7k$ vs $103.3k$ Txns/s). However, contention tanks the performance due to cascading aborts across batches. This motivates us to address the design challenge of sustaining the high performance of "+RU" in the face of low to moderate contention.

2.3 Our Approach

We develop Hackwrench to exploit caching and batching more effectively. To enable batched validation and commit, we propose a *two-tier commit* protocol in which transactions are first checked locally for serializability violations before being batched together and validated globally. At the core of Hackwrench is the mechanism *repair through fine-grained re-execution*, which can significantly lower the cost of invalidated transactions compared to wholesale aborts.

Tiered commit. We separate the usual monolithic commit protocol into two tiers (stages). In the first stage, referred to as "local commit", each node uses traditional local synchronization (e.g., 2PL) to execute transactions individually and ensure their serializability within a node. In the second stage, referred to as "global commit", each node groups together a batch of locally committed transactions and communicates with (multiple) data servers to validate the batch's read set and persist its write set.

Separating commits into two tiers allows us to handle intra-node conflicts using inexpensive local synchronization. More importantly, a locally committed transaction makes its writes visible to other transactions running on the same node, so they do not block waiting for the transaction's distributed global commit. As a result, we are able to batch together dependent transactions. Otherwise, we would be restricted to only batching together transactions with non-overlapping data access, which can seriously constrain throughput when there are only a limited number of such concurrent transactions.

Repair via fine-grained re-execution. Caching increases the chances of aborts due to stale reads. Tiered commit makes this situation worse because any transaction that has observed writes from an aborted transaction must also be aborted. To mitigate the cost of aborts, we need a more efficient solution than aborting and retrying a batch of transactions. Our insight is that it is cheaper to repair a batch of transactions by selectively re-executing only those operations affected by stale or invalid reads.

We implement such repair by representing transactions using static dataflow graphs so that the dependencies between operations are made explicit. Additionally, dependencies across different transactions are dynamically tracked through the tuples that they access. We illustrate the main idea of repair using an example. Figure 1 shows the dependencies among a batch of two locally committed transactions, T_1 and T_2, which access three data tuples A, B, C. We assume the version of a tuple is represented by its last writer transaction. Since a locally committed transaction exposes its writes to other transactions executing on the same database node, there exist implicit dependencies across transactions within a batch, as exemplified by the edges $W_3 \rightarrow R_6$ and $W_4 \rightarrow R_7$.

During the global commit, the read set of a batch is validated at relevant data servers. A read can be invalidated if its version does not match the tuple's current version due to conflicts or the stale cache. With the aid of the dataflow graph, we can repair the damage of the invalidated read by precisely identifying the subset of operations that need to be re-executed. In Figure 1, the read set consists of tuples A, B, C, all with version T_0. Suppose the read of tuple C (version T_0) fails its validation because the version has been changed to T_4, then operations R_2, W_4 of T_1 and R_7, W_{10} of T_2 must be re-executed using the new version of tuple C while all other operations are unaffected.

The idea of fine-grained re-execution is inspired by transaction healing and repair [18, 58], but differs in several important aspects. First, since repairing is done to a batch instead of an individual transaction, we need to track operation dependencies among different transactions in a batch. Second, we rely on explicit dataflow graphs to expose operation dependencies instead of static analysis [18] because the latter lacks precision.

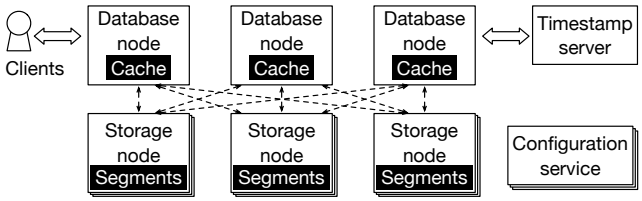


Figure 2: The architecture of Hackwrench. Dashed lines indicate communication links and stacked boxes indicate replicas.

3 Hackwrench Design

System overview. Figure 2 depicts Hackwrench's architecture. Hackwrench consists of three main components: a set of database nodes, a set of storage nodes, and a timestamp server. Database nodes execute transactions and coordinate their commits; storage nodes store data and validate transactions. As Hackwrench supports replication, we use the term *logical* storage node to refer to a group of *physical* storage nodes that replicate the same data (the default replication level is 3). Reading from (or writing to) a logical storage node requires contacting the read quorum (or write quorum) of its constituent physical storage nodes [53]. The size of the read/write quorum is configurable and must ensure non-empty quorum intersection. Hackwrench's tiered commit protocol guarantees strict serializability. It uses the timestamp server to ensure a consistent ordering of concurrent global commits from different database nodes.

For fault tolerance, Hackwrench relies on a Paxos-replicated configuration service to keep consensus on the current system configuration (aka view) which includes the identity of the timestamp server as well as the mapping of each data partition to its logical storage node. Each view is identified with a unique view number which is increased sequentially. RPC requests are attached with the view number of the system configuration known to the sender. The timestamp server and storage nodes reject requests whose view numbers do not match theirs. Such use of a configuration service is similar to that done in other distributed storage systems [13, 40, 44].

3.1 Data Organization and Caching

Hackwrench partitions data into segments, each of which contains a set of versioned key-value tuples belonging to a table. Users can designate a subset of the table's primary key columns as the partition key for each table. Each data segment is stored at a logical storage node. The configuration service maintains the mapping from each data segment to its logical storage node, which is cached by all the database nodes. The version of a tuple consists of a 63-bit unique ID of the last transaction that modifies that tuple and one "repaired" bit, which is needed to ensure that the writes of a re-executed transaction have versions different than those of its original execution.

Each database node keeps a large in-memory cache. Caching is done at the tuple granularity. In the face of a cache miss, a database node reads the tuple from the corresponding logical storage node. The cache is kept up-to-date asynchronously with no freshness guarantee.

3.2 Transaction Execution and Local Commit

In Hackwrench, transactions are represented as stored procedures. For OLTP workloads, stored procedures are commonly used for performance acceleration [45]. Unlike other systems [40, 42, 49, 52, 56, 59] that use C++-based stored procedures, Hackwrench provides a dataflow-based programming abstraction for users to write store procedures. Our dataflow APIs are inspired by Tensorflow [2], except that instead of supporting tensor operators, our API supports primitive operators on different tuple column types, including integers, strings, and floats, as well operators for reading and writing tuples in the database. All operators are deterministic. Their outputs are only dependent on their input, except for database reads, which depend on the current cached or database state. With our API, each transaction is represented by a static dataflow graph. In the actual implementation, we store one copy of the dataflow graph for a given transaction type at each node.

In Hackwrench's tiered commit protocol, concurrency control for a transaction is decomposed into two parts: local commit for resolving local conflicts within the same database node, and global commit for handling remote conflicts across different database nodes. The global commit protocol is discussed later in § 3.3.

Execution and local commit. To execute a transaction, a database node reads from its data cache whenever possible and buffers writes locally. It uses two-phase locking [26] (with NO_WAIT for deadlock prevention) to ensure strictly serializable execution. Upon finishing, the database node commits a transaction locally: it directly applies the transaction's writes to the database cache, making the writes visible to other transactions on the same database node. Locally committed transactions are then appended to one local queue of each database node, waiting for batched global commit. By releasing the locks held for 2PL after the transaction is pushed into the queue, we ensure that the order of transactions in the queue corresponds to their local commit order on the database node.

For performance's sake, it is crucial to expose a transaction's writes upon local commit. The alternative, i.e., holding locks during global commit , can seriously damage the system throughput because other transactions could be blocked from execution and local commit. However, there is a downside: if transaction T is aborted later during global commit, any transactions that have read T's uncommitted writes must also be aborted. This cascading effect can cause significant abort overhead, which Hackwrench seeks to mitigate using repair via fine-grained re-execution. To do so, Hackwrench needs to track the dependencies of operations among locally-committed transactions; if a read operation's output changes during validation, then all its dependent operations need to be re-executed as part of the repair. In fact, transactions are executed according to their dataflow graphs, which contain operation dependencies within the transactions. The details of runtime dependency tracking are described in Section 4.

3.3 Global Commit

Hackwrench dequeues a batch of locally committed transactions and tries to globally commit them at the logical storage nodes responsible for the batch's read set/write set. At the high level, Hackwrench's global commit follows the spirit of two-phase commit (2PC) where a database node coordinates with the set of participating logical storage nodes to go through a *prepare* phase followed by *commit* phase. Similar to [4], the *prepare* phase validates the batch's read set and persists its write set for crash recovery. We introduce two crucial variations to 2PC [4]. First, Hackwrench relies on a centralized timestamp server to assign the batch of transactions a consistent commit ordering. Second, instead of aborting the whole batch upon detection of conflict, Hackwrench repairs conflicted transactions. Next, we describe the commit timestamp assignment and the global commit procedure

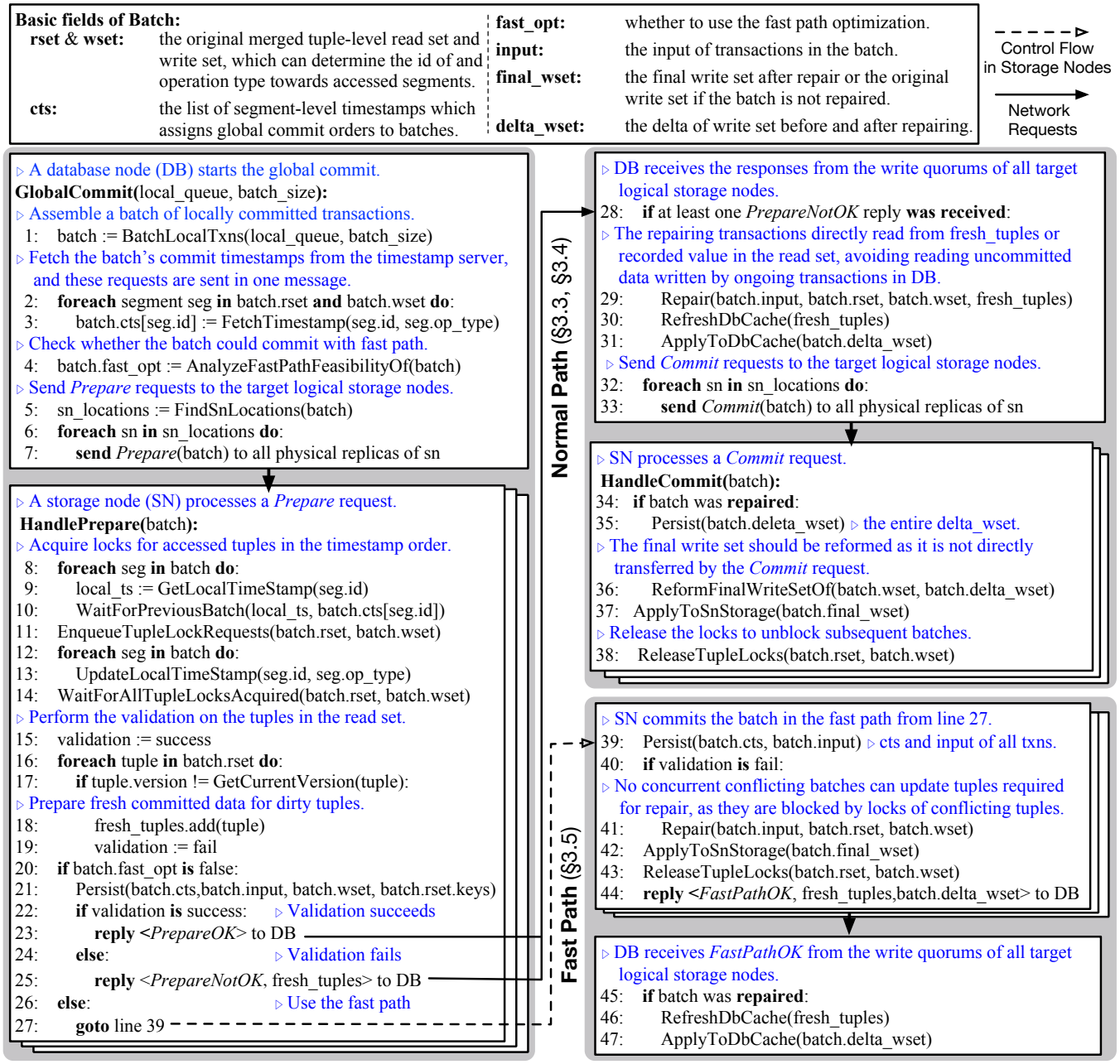


Figure 3: The algorithm for global commit. Procedures in stacked boxes are executed on the storage node.

without validation failure. Repair is discussed in § 3.4. Finally, we propose an optimization that enables certain types of transactions to be repaired efficiently at the logical storage nodes (§ 3.5).

Commit timestamp assignment. The goal of the centralized commit timestamp assignment is to ensure that all logical storage nodes agree on a consistent ordering when handling *conflicting* batches of transactions. Suppose transaction T_1 and T_2 have conflicting accesses on tuples x, y which are stored at nodes sn_x and sn_y, respectively. With commit timestamping, we aim to guarantee that both servers sn_x and sn_y will validate and commit T_1 and T_2 in the same order.

One design is to assign a total order to commit timestamps. This strategy is straightforward and adopted by many systems, e.g. FoundationDB [65], Spanner [16], and Deterministic DB [1, 49]. We instead assign commit timestamps using partial ordering, which can also ensure the consistent ordering of conflicting transactions, with the added advantage that non-conflicting batches will not unnecessarily block each other.

In Hackwrench, the timestamp server maintains a counter for each data segment. The per-segment counter is represented as a pair: <*seq*, *readers*>, where *seq* tracks the number of batches that have written the segment and *readers* tracks the number of batches that have read

the segment's latest write. When requesting a commit timestamp for a batch of transactions, the database node submits the list of segments in the batch's read set and write set to the timestamp server. The timestamp server locally locks all segments, reads each segment's current counter value into the commit timestamp, increments each write segment's *seq* field while zeroing its *readers* field, and increments each read segment's *readers* field while leaving its *seq* field unchanged.

Storage nodes handle global commits according to the partial ordering of commit timestamps. Suppose a batch with segment s has commit timestamp $CTS[s]$ =<*seq*,*readers*>. If s is in the read set, the storage node must wait for the arrival of the batch that has written to s, aka batch with timestamp $CTS[s]$=<*seq*,0>; If s is in the write set, the storage node must wait for the arrival of all *readers* batches that have read s, aka batches with timestamps $CTS[s]$=<*seq*, i>, where $0 \leq i \leq readers$.

Our scheme ensures that conflicting batches are handled in a consistent order at storage nodes. We note that while commit timestamps are coarse-grained at segment level, storage nodes still use tuple-level locking during validation to minimize false blocking and false conflicts. Because timestamps are coarse-grained and assigned to a batch of transactions, a single timestamp server can support high transaction throughput (§ 5.6, Figure 15).

Batched global commit. Hackwrench's global commit process works at the batch granularity. The pseudocode of the commit protocol is shown in Figure 3. To start, the database node dequeues a batch of transactions from its local queue (Line 1), with the local commit order of transactions preserved (§ 3.2). The batch's read set and write set are merged from its transactions' read set and write set, with careful deduplication. The batch's read and write set determine the set of participating logical storage nodes involved in the global commit.

After a batch is assembled, the database node fetches a commit timestamp from the timestamp server (Line 3). The timestamp server handles requests from the same database node in order, to ensure that commit timestamps are consistent with the local commit order of batches.

After obtaining the batch's commit timestamp, the database node proceeds to the *prepare* phase. It prepares the batch's read set for validation and the write set for persistence as redo logs. Let SN_{read} (or SN_{write}) denote the set of logical storage nodes managing the segments for the read set (or write set). The database node sends Prepare requests in parallel to all storage nodes in SN_{read} and SN_{write} (Lines 6,7). The Prepare request contains the batch's commit timestamp, read set, write set, and transactions' inputs.

Upon receiving a batch's Prepare request, the storage node acquires tuple-level locks for the batch, following the commit timestamp's partial order. It first checks whether this request must wait for other batches to finish enqueuing their lock requests (Line 10), by comparing its responsible segments' current timestamps with those from the batch's commit timestamp. Once the waiting is over, the storage node enqueues a lock request in a FIFO queue for each tuple according to the batch's read set and write set (Line 11) and updates the accessed segments' timestamp (Lines 12,13), thereby allowing subsequent batches to enqueue their lock requests.

Once a batch's locks have all been acquired (Line 14), the storage node can validate the batch's read set (Lines 16,17). If a tuple's current version does not match the one in the read set, then the validation fails (Line 19). When a transaction's read depends on the write of other transactions in the same batch, the corresponding validation is skipped. No matter whether the validation succeeds or fails, the storage node persists the information contained in the Prepare request (Line 21). For the batch's read set, only keys are persisted.

If validation succeeds, the storage node replies with PrepareOK (Line 23). Once the database node receives replies from a *write quorum* of each participating logical storage node and all the replies are PrepareOK, it can enter the *commit* phase to notify clients and send the commit decision to storage nodes (Lines 32, 33). Finally, when a storage node receives the Commit request, it knows the corresponding batch has been committed, applies the batch's write set to local storage, and releases its locks (Lines 37, 38).

3.4 Transaction Repair

When validation fails (Line 19) due to conflicting transactions from different database nodes, the storage node sends back those tuples which have caused invalid reads in the PrepareNotOK reply (Line 18 and 25) to help the database node refreshes its local cache. Note that storage nodes do not release tuple-level locks at this point. After the database node receives replies from the write quorums of all participating logical storage nodes, the database node proceeds to repair the batch if *any* PrepareNotOK reply is received (Line 29). We take the example shown in Figure 1 and assume R_2 fails validation. The repair procedure *sequentially* processes every transaction in one batch as follows. It starts with the first transaction in the batch, T_1. Checking T_1's reads, the procedure detects that R_2 needs to be re-executed. It re-executes R_2 which results in the write W_4 to tuple C being repaired as well. The procedure then proceeds to the next transaction in the batch, T_2. Checking T_2's reads, the procedure detects the change of C, and re-executes the affected operations R_7 and W_{10}, resulting in a new final write to tuple C. After all the transactions are checked and repaired if necessary, the re-execution can finish and the database node notifies clients of the results. Finally, the database node refreshes its stale cache (Line 30) and then applies the delta of write set before and after repair ($delta_{wset}$) to its cache (Lines 31).

After completing re-execution, the database node sends the Commit requests to all participating storage nodes (Lines 32, 33). Instead of transferring the final write set ($final\_wset$), the database node saves network I/O by sending the delta of write set, which can be used to reconstruct the final write set at the storage node (Line 36). Then, the storage node applies the final write set to local storage and releases the batch's locks (Lines 37,38).

Our design guarantees that the repair will succeed, except for two scenarios. The first scenario is when a read/write operation changes its key during re-execution. For repair to be successful, tuple-level locks must be held during re-execution to protect the read and write set. However, as these locks are acquired by storage nodes during the original validation, they do not cover the changed keys. The second scenario is when a user-initiated abort is triggered during re-execution. In both cases, the database node aborts the corresponding transaction and replaces it with a special `NOP` transaction, which means its write set is nullified. Ideally, aborts during repair should be rare. There are no such aborts in many workloads because their transactions' read set and write set are not affected by execution [20, 21, 35, 36, 49, 50, 61].

3.5 Fast-Path Optimization

A common but restricted form of transaction is the so-called one-shot transaction [30, 45]. In our setting, a one-shot transaction is one whose dataflow graph be decomposed into subgraphs, each of which only accesses data within a single data partition and can execute and reach a commit decision independently. For one-shot transactions, we can optimize the global commit process to let each storage node independently commit a batch of transactions without coordinating with others (aka without waiting for the Commit request of 2PC). This is possible for one-shot transactions if we *offload repair from database nodes to storage nodes* so that each logical storage node can independently repair (its portion of) the batch successfully.

We use the example in Figure 1 to illustrate fast-path optimization for one-shot transactions. Let us assume tuples A, B belong to the same data partition stored at logical storage node sn_1, and tuple C belongs to a different partition stored at logical storage node sn_2. We can partition each transaction's dataflow graph into two pieces such that there are no dependencies between the subgraph accessing A/B and the subgraph accessing C. Therefore, this batch qualifies for fast path global commit. The database node sends Prepare requests to logical storage nodes. Suppose operation R_2 fails validation, instead of returning to and repairing at the database node, sn_2 directly repairs T_1's piece ($R_2 \rightarrow W_4$) and T_2's piece ($R_7 \rightarrow W_{10}$). As these two pieces only access tuple C and do not have dependencies with the pieces sent to sn_1, sn_2 can independently perform the repair and commit T_1 and T_2.

The pseudocode for the fast path is shown in the bottom right corner of Figure 3. In this case, every logical storage node persists the commit timestamp and input of *all* transactions in the batch (Line 39), which is needed for failure recovery (§3.6). The storage node repairs the batch locally if validation fails (Line 41), and commits without waiting for 2PC Commit requests (Lines 42-44). This can greatly improve commit throughput under contended workloads because the logical storage nodes can "pipeline" their handling of batches that conflict on the same segment, incurring no network delay. Finally, the logical storage node replies to the requesting database node with FastPathOK. Once the database node receives FastPathOK from the write quorums of all participating logical storage nodes, it can notify the clients. If the batch has been repaired, the database node then refreshes its cache with fresh tuples and the delta of write set (Lines 46-47).

3.6 Failure Recovery

Failure model. We assume that database nodes and the timestamp server can fail, but replicated logical storage nodes *do not* fail.

Recovering the database node failures. When recovering from database failures, we must complete (i.e., either commit or abort) all pending transactions from the failed database node. This is because unfinished transactions can impede progress by blocking conflicting transactions from other database nodes. Our design relies on a replicated configuration service for failure detection, and to initiate recovery. Upon detecting a database node failure, the configuration service notifies all storage nodes, which then stop processing Prepare and Commit requests from the failed node. Next, the configuration service appoints another database node to be a (recovery) coordinator. The coordinator contacts the timestamp server to identify pending transaction batches from the failed node, and the logical storage nodes to determine what information has been persisted for each batch. The coordinator uses this information to decide whether each pending batch should be committed (because sufficient information is available to do so), or aborted, and we describe both cases below. Recovery is complete once all pending batches have been either committed or aborted.

A pending batch is aborted if any of its participating logical storage nodes have not persisted its Prepare request. This is because as an optimization, we only include the part of the batch that is relevant to a storage node in its Prepare message, and must combine these during recovery. Similarly, we commit a pending batch if *all* participating logical storage nodes have persisted its Prepare requests and *no repair* is required. Handling batches that require repair is more complex, because it requires recomputing the final write set, and consequently, our handling depends on the state at logical storage nodes.

If the recovering batch has not been committed at any logical storage node, we use the Prepare requests, which contain the commit timestamp, transaction inputs, and read set and write set keys to recompute the final write set. To do so, we re-execute the transaction using the tuples stored at the storage nodes, the timestamp, and the transaction input. Similar to the transaction repair logic, our recovery logic aborts transactions for which the set of keys in the original read or write set changes during re-execution.

On the other hand, if the batch was committed at a logical storage node, we must commit it at other storage nodes. However, we cannot simply re-execute the transaction: tuples in any logical storage node that committed the transaction might have been updated subsequently by batches with later timestamps. To address this problem, Hackwrench persists the *entire* delta of a batch's write set and not just the part relevant to the storage node before committing the batch at that node. During recovery, the coordinator reads this delta write set from a logical storage node that committed the transaction, combines it with the write set from the persisted Prepare request to construct the final write set, and uses this write set to commit the batch.

Transactions processed with the fast-path optimization are handled slightly differently because storage nodes independently make commit decisions for them. Our recovery procedure for these transactions depends on two design choices: first, the Prepare request sent to participating logical storage nodes contains timestamps and transaction inputs for all transactions in the batch; and second, transaction inputs suffice to re-execute the one-shot transactions contained in a batch that uses the fast-path optimization. Therefore, recovering these batches requires the recovery coordinator to re-send the Prepare request to all participating logical storage nodes.

Recovering from timestamp server failures. We also rely on the replicated configuration to detect timestamp server failures. However, handling this requires global quiescence and a view change. To do so, the configuration service informs all storage nodes to stop processing new transactions and finish all pending ones in the current view, using the scheme for recovering failed database nodes if necessary. Once this is finished, the configuration service can install at all storage nodes the new view containing the new timestamp server. The new timestamp server assigns timestamps starting with the initial value. Storage nodes only wait for batches whose timestamps belong to the current view. We believe one can use view changes to add or remove storage nodes, however, our prototype does not support such changes.

3.7 Correctness

Next, we provide a proof sketch showing that Hackwrench implements strict serializability. The proof follows Adya's formulation for transaction isolation [3], which we briefly review below. A *schedule* is a sequence of operations performed by transactions. A schedule s is *serializable* if there exists a schedule s' where committed transactions are executed serially and produce the same results as in s. A schedule s is *strictly serializable* if it is serializable and such s' preserves the real-time order: for any transaction T_1 and T_2, if T_1 commits before T_2 begins, then T_1 is serialized before T_2 in s'. Transactions have *dependencies*: Transaction T_2 *read-depends* on transaction T_1 if some operation o_2 in T_2 reads a tuple version written by some operation o_1 in T_1; T_2 *write-depends* on T_1 if o_2 overwrites a tuple version written by o_1; T_2 *anti-depends* on T_1 if o_2 overwrites a tuple version read by o_1. Transaction T_2 *depends* on transaction T_1 if T_2 read-, write-, or anti-depends on T_1. Further, for two commit timestamps cts_1 and cts_2 of two batches, we say $cts_1 < cts_2$ iff there exists at least one segment s they have in common and $cts_1[s]$ is before $cts_2[s]$ in the timestamp order.

Definition 1 (Effective Operation). An operation o of a committed transaction T is *effective* if T has not been repaired or, during repair, o is not discarded due to re-execution or control flow changes.

Definition 2 (Repair Direct Serialization Graph). A *repair direct serialization graph* (RDSG) is a directed graph where every vertex represents a committed transaction, and there exists one directed edge $T_1 \rightarrow T_2$ iff T_2 depends on T_1 and the operations establishing the dependency are effective.

Lemma 1. *For any transaction T_1 and T_2 that are executed on different database nodes, if $T_1 \rightarrow T_2$ in the RDSG, then the batch B_1 that contains T_1 globally commits before the batch B_2 that contains T_2.*

PROOF SKETCH. There are two cases.

Case 1. T_2 read- or write-depends on T_1. In this case, T_2 must have accessed a tuple version written by T_1 and this tuple version must be committed. If the tuple version is not committed, T_2 cannot have accessed such a version as T_1, and T_2 are executed on different database nodes that do not share the same cache. Therefore, T_1 must globally commit before T_2, and due to the dependency (aka conflicts) between T_1 and T_2, B_1 must globally commit before B_2.

Case 2. T_2 anti-depends on T_1. Let o_1 and o_2 respectively denote the effective operations from T_1 and T_2 that establish the anti-dependency. B_1 must have performed validation before B_2 commits at the storage node containing the segment o_1 and o_2 access. Otherwise, B_1 would fail at the validation of o_1, and o_1 would be re-executed or even be discarded by control flow changes, which would disqualify o_1 as an effective operation. As storage nodes handle conflicting batches sequentially WRT timestamp ordering, B_1 must globally commit before B_2. □

Lemma 2. *For any committed transactions T_1 (from batch B_1 with commit timestamp cts_1) and T_2 (from batch B_2 with commit timestamp cts_2) such that $B_1 \neq B_2$, if $T_1 \rightarrow T_2$ in the RDSG, then $cts_1 < cts_2$.*

PROOF SKETCH. Let o_1 and o_2 denote the effective operations from T_1 and T_2 that establish the dependency, respectively.

Case 1. T_1 and T_2 are executed on the same database node, and B_1 is thus serialized before B_2. As the timestamp server never reorders requests from the same database, $cts_1 < cts_2$ must hold.

Case 2. T_1 and T_2 are executed on different database nodes. By Lemma 1, B_1 globally commits before B_2. As the storage node handles B_1 and B_2 in timestamp order, the commit order between B_1 and B_2 implies that $cts_1 < cts_2$. □

Lemma 3. *Hackwrench's RDSGs are acyclic.*

PROOF SKETCH. We prove the lemma via contradiction, assuming that there exists a valid schedule in Hackwrench whose RDSG contains a cycle denoted as $T_1 \rightarrow \cdots \rightarrow T_n \rightarrow T_1$. T_1 and T_n must belong to different batches as transactions in the same batch are totally ordered. Let B_1 and B_n denote the batches that contain T_1 and T_n, whose commit timestamps are cts_1 and cts_n, respectively. By Lemma 2, $cts_1 < cts_n$ as $T_1 \rightarrow T_n$ and $cts_n < cts_1$ as $T_n \rightarrow T_1$. However, the relationship between cts_1 and cts_n contradicts the fact that the timestamp server ensures a partial order among the commit timestamps. □

Theorem 1. *Hackwrench preserves serializability.*

PROOF SKETCH. We first show that for any valid schedule s_r with repair in Hackwrench, there exists a valid schedule s without repair that produces the same results as in s_r. The key is that, though the order of operations for each transaction may not obey the programming order due to repair, the dependency between operations always preserves the relative order among conflicting operations in a transaction, as dependent operations' re-execution would be triggered by the re-execution of the operation they depend on. We then show that for any valid schedule s without repair in Hackwrench, there exists a valid serial schedule of committed transactions that produce the same results as in s. By Lemma 3, RDSGs derived from s are acyclic, thus allowing producing serial schedules by topological sorting. The validity of such serial schedules follows directly from [10]. Therefore, by definition, Hackwrench preserves serializability. □

Lemma 4. *For any transaction T_1 and T_2, if T_1 globally commits before T_2 begins, then there is no path from T_2 to T_1 in the RDSG.*

PROOF SKETCH. We prove the lemma by contradiction, assuming that there exists a path $p = (T_2, ..., T_1)$ in the RDSG. T_1 and T_2 must be executed in different batches because transactions in a batch commit simultaneously by design and T_2 could not have begun before T_1 globally commits. T_1 must wait for T_2 to global commit before T_1 is validated, because for any edge $T_i \rightarrow T_j$ in p and T_i and T_j are in different batches, T_j must wait for T_i to global commit before T_j is validated and committed. This implies that T_2 begins before T_1 globally commits, causing contradiction. □

Theorem 2. *Hackwrench preserves strict serializability.*

PROOF SKETCH. We prove it by construction. By adding edge (T_i, T_j), where T_i globally commits before T_j begins, to any RDSG g, we could obtain an acyclic graph g'. The argument for the acyclicness of g' is similar to [41] based on Lemma 4, which we skip for brevity. Following the same reasoning in the proof sketch for Theorem 1, topologically sorting g' gives a valid serial schedule WRT the real time order. By definition, Hackwrench preserves strict serializability. □

4 Implementation

This section describes the implementation details of Hackwrench. All nodes assign one worker thread per core, which executes an event loop to process network messages, transaction logic, etc. We use 64MB segments and currently store data and metadata in memory.

Batch splitting. A naive approach to forming batches is to combine all pending transactions into a batch. However, this approach has two drawbacks: 1) it limits parallelism because non-conflicting

transactions execute sequentially, and 2) repairs to a transaction can unnecessarily delay commit for a non-conflicting transaction.

Therefore, Hackwrench splits a batch into non-conflicting sub-batches to improve parallelism. Specifically, after a batch is formed, the database node constructs an undirected graph with transactions as nodes and edges connecting transactions accessing the same segment. A connected component in this graph represents a set of conflicting transactions, and we merge these into a sub-batch. Different sub-batches in a batch access different segments, and can be validated, repaired and committed concurrently.

Dependency tracking. Hackwrench needs to track dependencies among transaction data accesses (the dashed lines in Figure 1). To do so, for any read Op_r, Hackwrench stores the <key,$version$> pair of each accessed tuple. In this case, the $version$ indicates the last transaction T_w that wrote to key's tuple, it implicitly records Op_r's dependency on T_w. During execution, Hackwrench does not record any write dependencies, since these can be inferred from Hackwrench's dataflow API statically. Tracking dependency adds an overhead of 16 bytes per tuple read by a transaction.

Fine-grained re-execution. An intuitive approach for fine-grained re-execution is constructing a large operation-level dependency graph for all transactions in one batch and repairing transactions accordingly. However, this approach introduces runtime overhead for dependency graph construction and traversal. Hackwrench uses a simple but efficient way instead. Thanks to the local queue on each database node, transactions within the same batch are totally ordered. Hackwrench repairs transactions sequentially in the total order. To repair one transaction, Hackwrench re-executes it according to its static dataflow graph. For each read, Hackwrench compares the recorded <key,$version$> pair with the current key and tuple version. If such metadata changes, then the read is repaired. Writes are repaired if one of the reads they depend on is repaired, according to the dependencies encoded in the dataflow graph. When a write is repaired, it assigns the tuple a new version containing the repaired transaction's ID and sets the repair bit to true. The complexity of the above repair procedure is O(N), where N is the number of read operations in one batch.

5 Evaluation

5.1 Experiment Setup

5.1.1 Comparison targets. We evaluate Hackwrench by comparing to five systems: an OCC implementation, FoundationDB [65], COCO [40], Calvin [49], and Sundial [63]. Hackwrench, OCC, and FoundationDB use remote storage. COCO, Calvin, and Sundial use co-located storage. For all systems, we store data in memory. We use three-way replication for all systems other than Sundial. We used the original implementation from GitHub [5, 38, 60, 62] for FoundationDB, COCO, Calvin and Sundial.

Hackwrench. Our default configuration enables fast-path optimization. For comparison, Hackwrench-nofast in the evaluation refers to a version where the fast-path optimization is disabled. In both cases, we set the read quorum to one and the write quorum to three.

OCC. The OCC system we compare against modifies Hackwrench's code, and uses tiered commits. Our implementation uses 1) two-phase locking [26] (with NO_WAIT) to handle local conflicts, 2) optimistic concurrency control [31] to resolve remote conflicts, and 3) two-phase commit for global committing. By default we neither cache data, nor batch transactions. The OCC+Caching results we present represent an OCC configuration where caching is enabled. Note, this OCC implementation uses tiered commit, and thus differs from the naive-OCC implementation used for Table 1. However, caching, batching, and uncommitted reads have similar impact on their performance.

FoundationDB (FDB). Similar to Hackwrench, FDB is an open-source system that uses remote storage. FDB uses a centralized sequencer to assign totally-ordered commit timestamps and performs OCC-style validation. Unlike FDB, Hackwrench timestamps are partially ordered, allowing greeater parallelism. Unlike Hackwrench and OCC, FDB cannot cache data in database servers[3]. Each FDB component is implemented by a separate process, and we assign separate cores to each process. We partition processes across physical servers as follows: 1) Database servers, which run client processes that execute transaction logic, and one proxy process that acts as the transaction coordinator. 2) Storage servers, which run one logging process and data storage processes. Every two storage servers share one resolver process, which is used to validate transactions. 3) One configuration server, which runs the sequencer process that provides timestamps. For a fair comparison, we configure FDB to use its memory storage engine and tmpfs as persistent storage. For all the experiments, we use one storage server per database server because this configuration resulted in the best FDB performance.

Calvin. Calvin is a deterministic database that assigns transactions a commit order before execution. A single thread schedules these totally ordered transactions on each server, limiting parallelism. Calvin also batches transactions to improve performance and replicates each transaction for fault tolerance. Unlike Calvin, Hackwrench assigns commit order after transaction execution and thus has better concurrency.

COCO. COCO is a distributed database with OCC and 2PC. When committing transactions, it validates each transaction individually, and batches validated transactions together for replication. COCO does not batch transaction validation to avoid cascading aborts. By comparison, Hackwrench batches both transaction validation and replication because it uses fine-grained re-execution. We use COCO-colocated to denote measurements from the original implementation.

Sundial. Sundial is a distributed database with caching. It uses logical leases for cache management and reduces the probability of OCC validation fails. In contrast to Hackwrench, Sundial sends network messages for remote write operations to lock tuples, thus negating the benefits of caching. Hackwrench instead allows transactions to read uncommitted data and stale data from caches, and uses fine-grained re-execution for repair, thus avoiding this overhead. The Sundial implementation does not currently support replication. We use Sundial-colocated to denote measurements from the original implementation.

COCO and Sundial's design assumes that storage and transaction processing are co-located. Consequently, using their concurrency control protocols with remote storage introduces message overheads due to the need to validate data by comparing it to the current value in remote storage. These systems, unlike Hackwrench, don't implement tiered commits. Therefore, they must incur the message overhead even when the transaction is aborted due to conflicts with transactions executed by the same database node. We measure these overheads

[3]FoundationDB's documentation mentions "distributed caching," but we have confiemed with the developers that caching support is currently not implemented.

TPC-C Transaction Types	NewOrder			Payment			OrderStatus			Delivery			StockLevel		
Latencyes (ms)	P50	P90	P99	P50	P90	P99	P50	P90	P99	P50	P90	P99	P50	P90	P99
HackWrench(BS=1)	0.75	0.84	0.98	0.66	0.75	0.91	0.65	0.75	0.94	1.10	1.25	1.38	1.37	1.52	1.63
HackWrench(BS=50)	2.56	3.20	3.93	2.55	3.18	3.85	2.55	3.13	3.98	2.78	3.40	3.99	2.77	3.42	4.26
HackWrench-nofast(BS=1)	1.14	1.31	1.63	1.05	1.33	1.64	1.00	1.11	1.27	1.55	1.69	1.90	1.77	1.94	2.16
HackWrench-nofast(BS=50)	2.93	3.52	4.13	2.91	3.51	4.11	2.90	3.45	4.06	3.12	3.68	4.28	3.19	3.75	4.33
OCC	2.65	5.58	10.17	2.10	4.17	8.59	1.70	1.99	2.92	2.93	3.21	5.58	3.43	5.87	9.50
OCC+Caching	1.33	2.35	4.42	1.20	2.19	3.98	1.06	1.21	1.65	1.69	2.25	3.41	2.04	3.24	5.36
FDB	7.95	10.32	14.37	3.64	4.75	8.4	2.8	3.09	6.77	27.05	31.59	39.25	40.79	46.85	54.08

Table 2: Latencies in standard TPC-C ($r = 1\%$). "BS" stands for "batch size". The darker green/red lines have lower/higher latencies.

using modified implementations of both (COCO-remote and Sundial-remote in the graphs) that use remote storage. Our modification forces any single-server transaction[4] in COCO or Sundial to do a 2PC procedure with one remote server. This allows us to simulate a case where the transactions use a local cache (no messages are sent for reads) but are committed at a remote storage server.

5.1.2 Benchmarks and workloads.

We use two benchmarks for our evaluation: (a) The FoundationDB's **FDB-Micro** microbenchmark [65]; and (b) the **TPC-C** benchmark [51].

FDB-Micro. This benchmark has 214M tuples in total, each of which consists of an 8-byte key and a 24-byte value. The tuples are partitioned evenly across storage nodes, and database nodes, according to their keys. The workload is a mix of 80% read-only transactions with 20% read-write transactions. Each read-only transaction reads ten tuples. Each read-write transaction reads five tuples and updates another five tuples. In this workload, a local transaction is one where the database only accesses tuples in its partition, while a distributed transaction is one where an operation accesses a tuple in a remote database node's partition. The parameter d dictates the percentage of distribution transactions, the rest are local transactions. The tuples are accessed following uniform distribution (low contention) or Zipfian distribution [24] with $\theta = 0.99$ (high contention).

TPC-C. By default, we run the standard TPC-C benchmark with five different types of transactions. In the standard benchmark, when a NewOrder transaction updates the Stock tuple, there is a 1% probability that the tuple belongs to a remote warehouse. The higher the remote access probability (r), the more remote distributed NewOrder transactions are. Thus, we vary r to see how Hackwrench performs. We use warehouse IDs to determine the database node that executes a transaction. In our workload, each database node is associated with 8 warehouses.

5.1.3 Setup.

We ran all experiments on a cluster of Amazon EC2 `m5.2xlarge` instances, each with eight 3.1 GHz virtual CPUs, 32GB of RAM, and 10Gbps network bandwidth. We configure the number of database and storage nodes to saturate database nodes' processors. By default, we use 6/8 database nodes, 1/1 timestamp server, and 4/6 logical storage nodes for TPC-C/FDB-Micro. When enabling replication, each logical storage node consists of 3 physical replicas.

5.2 Comparison to Systems with Remote Storage

Throughput. Figure 4 shows that, on TPC-C, Hackwrench and Hackwrench-nofast outperform OCC+Caching by up to 9.0× and 6.8×, and outperform FDB by up to 35.8× and 27.8×. They perform better than OCC+Caching due to batched transaction commit and allowing uncommitted read. 88% of TPC-C transactions (45% NewOrder and 43% Payment) read or write the same tuple in each warehouse and contention on this per-warehouse tuple limits parallel execution. While Hackwrench and Hackwrench-nofast allow transactions to access tuples that are not globally committed, improving parallelism. In addition, they repair transactions instead of aborting them. For Hackwrench, the fast-path optimization enables the storage node to commit the batches without waiting for network messages. FDB performs worst because it does not cache data and issues a remote request for each read. Additionally, FDB does not allow its client process to batch transactions and thus limits parallelism. Except for FDB, the performance of all systems drops as the remote access probability increases. It is because FDB does not use caching (which reduces the chances of stale reads) and relatively low throughput.

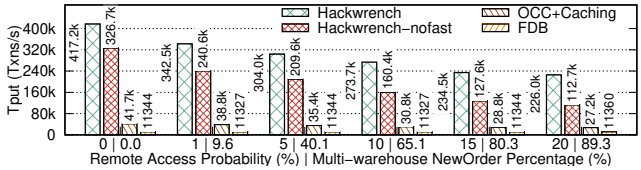


Figure 4: Comparison of systems with remote storage on TPC-C. The multi-warehouse NewOrder transaction percentage is calculated from the remote access probability.

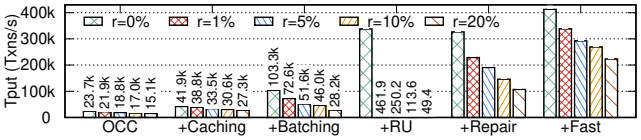


Figure 5: Design breakdown on TPC-C. The designs are applied incrementally based on OCC. "+Caching": adding local caches. "+Batching": supporting batching. "+RU": permitting uncommitted reads across batches. "+Repair": using repair. "+Fast": enabling fast-path optimization.

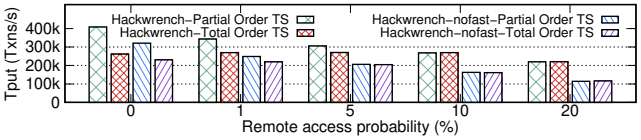


Figure 6: Partial order and total order timestamps comparison.

Latency. Table 2 shows the zero-load latencies of different systems on standard TPC-C. When transactions are batched, Hackwrench and Hackwrench-nofast exhibit a little higher latency than OCC+Caching. This is due to time spent forming batches, and indeed, when batch size is 1, Hackwrench and Hackwrench-nofast have lower latency than OCC+Caching. As expected, we find that FDB has the highest latency because it does not cache data. Although OCC also does not use caching, it reads multiple tuples in one network round trip and thus has lower latency than FDB.

5.3 Factor analysis

Design breakdown. Figure 5 uses TPC-C to measure the impact of each of Hackwrench's design decisions. We use OCC as the baseline.

[4] A single-server transaction's execution and commit can be completed in one server without waiting for any network delay.

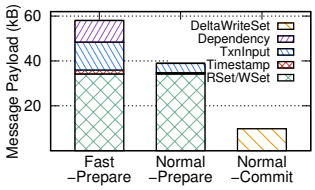


Figure 7: Message payloads.

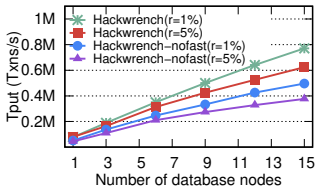


Figure 8: Scalability (TPC-C).

Caching improves performance by 76.7% ~ 80.5%. Batching and uncommitted reads ("+RU") improve performance by 7.1× for the $r = 0\%$ case, but significantly degrade performance for $r > 1\%$, because increasing r introduces remote conflicts, leading to aborts. Furthermore, this configuration is also susceptible to batch-level aborts and cascading aborts, where one aborted transaction in a batch can lead to future batches being aborted, further degrading performance.

The timestamp server and database-side repair mitigate these effects. Repair largely mitigates overheads from cascading aborts, allowing the resulting system ("+Repair", or Hackwrench-nofast) to fully exploit the performance benefits of the previous design choices. Finally, the fast-path optimization ("+Fast", or Hackwrench) allows the storage nodes to commit batches without waiting for network round-trip, improving performance by 26.5% ~ 108.9% compared to "+Repair". These improvements grow with larger r.

Cost of tracking and using dependency. Hackwrench uses runtime dependency tracking for fine-grained re-execution. We measured the space utilization for Hackwrench's dependency tracking on standard TPC-C. An average TPC-C transaction uses 586 bytes for dependency tracking and spends 4.1μs checking for repairs.

Partial order vs total order timestamp. Figure 6 uses TPC-C to measure the impact of Hackwrench's partially ordered timestamps against FDB's totally ordered timestamp [65]. Partial ordering improves performance significantly when remote access probabilities are low. This is because a design with total order requires database nodes to broadcast any transaction batch to all logical storage nodes for progress, on the other hand with partial ordering we only need to send transaction batches to relevant storage nodes, reducing overheads.

Network I/O cost. Figure 7 shows the breakdown of Hackwrench's average message (or request) payloads on standard TPC-C. The Commit requests contain the *entire* delta of write set (§ 3.6). Note that the fast path does not send Commit requests, because storage nodes can commit fast-path transactions without coordination. The average payload for Prepare requests increases from 38.9kB to 58.0kB (by 49.1%) when fast-path optimization is used. Read set and write set payloads remain unchanged, while commit timestamps and transaction input payloads increase from 0.5kB and 4.1kB to 1.7kB and 12.4kB, respectively. This is because each participating storage node must replicate *all* commit timestamps and transaction inputs in a batch, not just the relevant portions. As storage nodes repair transactions in the fast path, the tracked dependency (9.6kB) must be sent.

Latency for determining the commit order. Hackwrench requires a network round-trip to determine commit order, in TPC-C, we measured this to be 0.28ms on average.

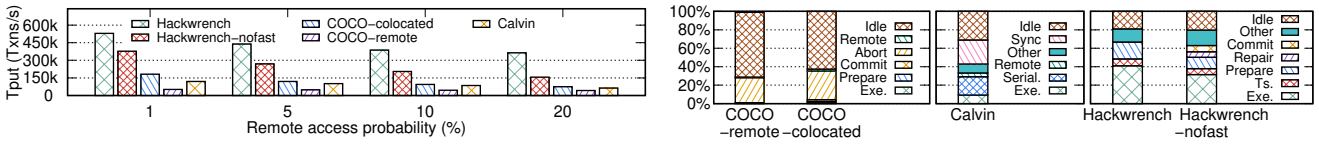
5.4 Scalability

Next, we evaluate Hackwrench's and Hackwrench-nofast's scalability on standard TPC-C. Our results in Figure 8 show that both are scalable: Hackwrench's 15-database throughput is 771.4k Txns/s (for $r = 1\%$) and 624.2k Txns/s (for $r = 5\%$), achieving 10.0× and 7.8× the throughput of a single database node which does not incur any cross-node conflict. Meanwhile, Hackwrench-nofast's 15-database throughput is 496.0k Txns/s (for $r = 1\%$) and 376.8k Txns/s (for $r = 5\%$), achieving 9.7× and 8.5× the single-database throughput. In our experiment, we found that because we use segment-level timestamps and batching, the timestamp server was not a scalability bottleneck. We evaluate timestamp server peak throughput in § 5.6.

5.5 Comparison to Systems with Co-located Storage

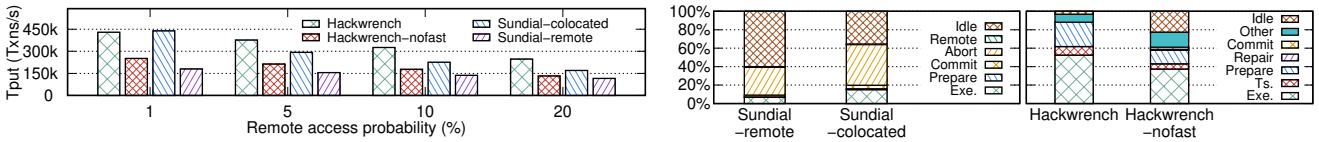
The COCO implementation only supports TPC-C NewOrder and Payment transactions [40], therefore we use a mixture of 50% NewOrder and 50% Payment transactions to compare against COCO and Calvin. We ran all systems on 19 m5.2xlarge instances and used 48 warehouses. Our results in Figure 9(a) show that Hackwrench and Hackwrench-nofast outperform the other systems because they use batching and uncommitted reads, and can thus support a larger number of ongoing transactions. In Table 3 we provide a breakdown of all systems' throughput, and CPU time in Figure 9(b). In these results, a 'thread' refers to threads used for transaction execution. Hackwrench and Hackwrench-nofast have higher per-thread throughput than both COCO variants. Further, COCO-remote and COCO-colocated spent only 0.36% and 1.95% of CPU time executing transactions, and the CPUs idle for the majority of the time. This is because, COCO does not batch validation and uses one transaction per thread, and COCO's threads sleep for one microsecond before retrying aborted transactions. By contrast, Hackwrench and Hackwrench-nofast use 41.0% and 31.5% of CPU time for execution, because batching and uncommitted reads allow us to avoid idling. Since Calvin executes each transaction on all server replicas, it has 7 ($\lceil 19/3 \rceil = 7$) (replicated) database nodes in total. It uses 9.17% CPU time for execution, and 19.9% and 26.7% of CPU time for data serialization and synchronization in its TPC-C benchmark implementation. The execution time ratio (between different systems) is not the same as the per-thread throughput ratio because transaction execution latency varies across systems. Hackwrench's throughput is 9.27× COCO-remote's, 3.9× COCO-colocated's, and 5.7× Calvin's throughput.

Our comparison with Sundial uses the default TPC-C mix, but disables replication. In this case, all systems use 11 m5.2xlarge instances and 48 warehouses. Our results in Figure 10(a) show that Sundial-colocated has the highest throughput for $r = 1\%$ because in this case 77.3% of the transactions are single-server transactions that benefit from its co-located architecture. However, because Sundial does not use batching, uncommitted reads or TPC-C's one-shot property, its performance declines quicker than Hackwrench as r increases. Sundial outperforms Hackwrench-nofast for $r = 20\%$ as 53.4% transactions are single-server. On the other hand, Hackwrench-nofast outperforms Sundial-remote for all r. Table 4 and Figure 10(b) show detailed throughput and CPU breakdowns. Like COCO, Sundial also allows one ongoing transaction on each thread and sleeps before retrying aborted transactions, thus leading to high CPU idle times. For transaction execution, Hackwrench and Hackwrench-nofast consume 52.4% and 37.9% of CPU time, respectively. By comparison, Sundial-remote and Sundial-colocated consume 7.0% and 15.1%.



(a) Throughput (b) Time breakdown for COCO, Calvin and Hackwrench

Figure 9: Performance on 50% NewOrder and 50% Payment TPC-C (19 nodes). "Exe.": transaction local execution. "Prepare" and "Commit": handling *prepare* and *commit* phases. "Abort": abort penalty. "Remote": serving remote requests. "Idle": idle time. "Serial.": data serialization. "Other": system logic such as batch formation and garbage collection. "Sync": synchronization in benchmark implementation. "TS": getting timestamps. "Repair": database-side repair.



(a) Throughput (b) Time breakdown for Sundial and Hackwrench without replica

Figure 10: Performance on standard TPC-C without replication (11 nodes). The legends have been explained in Figure 9.

Systems	Per-thread Tput	# threads / DB	# DB	Total Tput
Hackwrench	10708	8	6	514.0k
Hackwrench-nofast	7710	8	6	370.1k
COCO-colocated	1911	5	19	181.5k
COCO-remote	542	5	19	51.5k
Calvin	4079	4	7	114.2k

Table 3: Throughput breakdown for Figure 9(a), $r = 1\%$.

Systems	Per-thread Tput	# threads / DB	# DB	Total Tput
Hackwrench	8709	8	6	418.0k
Hackwrench-nofast	5628	8	6	270.1k
Sundial-colocated	6562	6	11	433.1k
Sundial-remote	2693	6	11	177.7k

Table 4: Throughput breakdown for Figure 10(a), $r = 1\%$.

5.6 FDB-Micro

Impact of Contention and Distributed Transactions. In Figure 11 we show results from FDB-Micro when in a low-contention workload and high-contention workload, while varying the percentage of distributed transactions. In the low-contention case, Hackwrench and Hackwrench-nofast have similar performance. COCO-colocated has the performance for small $d \leq 20\%$, because a significant fraction of transactions run on a single-server. COCO-colocated's performance drops as we increase d, and eventually is the same as that of COCO-remote. Under high contention, the performance gap between Hackwrench and Hackwrench-nofast increases because more batches are repaired ($4.4\% \sim 27.8\%$ v.s. $53.4\% \sim 99.7\%$). OCC+Caching throughput also drops significantly because of frequent aborts due to its use of 2PL with NO_WAIT for local conflicts, and policy of reading committed data. When $d = 100\%$, the average performance gap between COCO-colocated and COCO-remote is 56.6%, because COCO-remote must send network messages for local conflicts. Our FDB-Micro implementation for Calvin does not impose noticeable serialization and synchronization overheads. Instead, the single thread used by each server to schedule transactions and maintain a total order becomes the bottleneck in the low-contention setting. In the high-contention setting, Calvin achieves $92.5\% \sim 98.7\%$ of its low-contention throughput and has performance comparable to Hackwrench-nofast, because it orders transactions before execution, thus avoiding aborts due to contention. Finally, FDB's implementation of FDB-Micro retrieves all tuples

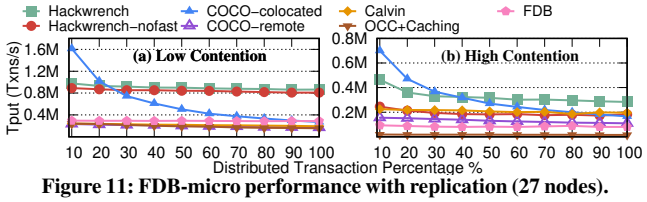


Figure 11: FDB-micro performance with replication (27 nodes).

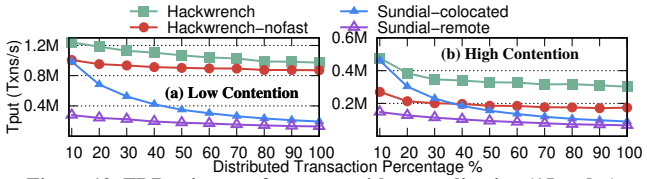


Figure 12: FDB-micro performance without replication (15 nodes).

in a single network round trip, uses 12 database nodes, and thus outperforms OCC+Caching (which uses 6 database nodes).

We evaluate Hackwrench and Sundial in Figure 12. When replication is disabled, Hackwrench's and Hackwrench-nofast's performance increases because CPU time for replication is saved and the number of database nodes remains unchanged. The performance trends of Sundial are similar to those of COCO.

Impact of Batching. Figure 13 varies Hackwrench's batch size from 1 to 160 on FDB-micro. In both contention levels, increasing batch sizes allow Hackwrench to support a higher offered load, though this does come at the cost of increasing latency. As the batch size increases, the zero-load latency also increases (e.g., from 0.56ms to 3.83ms under low contention). When contention is low, peak throughput improves by $1.95\times$ (from 362.3k to 1.06M Txns/s). In the high-contention case, the peak throughput improves by 81.7% (from 324.8k to 590.2k Txns/s), with a batch size of 40. The performance decreases with larger batch sizes, as the possibility of remote conflicts between batches also increases. We also measure cascading repairs due to batching. We find that under high contention, the average number of repaired transactions due to a remote grow from 2.23 repairs (batch size 1) to 441.4 (batch size 160).

Impact of Caching. Figure 14 controls the cache miss rate c for each data access on FDB-micro. As c decreases, Hackwrench's performance improves by up to $2.1\times$. Hackwrench and Hackwrench-nofast perform similarly as the contention is low. By contrast, OCC+Caching can only improve performance by up to 87.3%. These

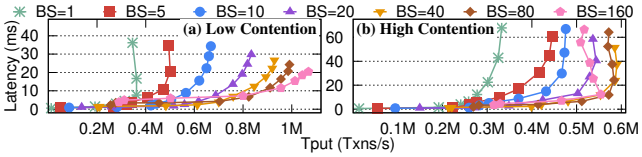


Figure 13: Impact of batching on FDB-micro (low contention, $d = 10\%$). "BS" stands for batch size.

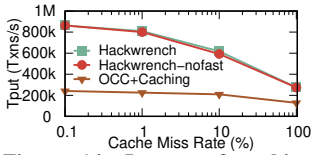


Figure 14: Impact of caching on FDB-micro (low contention, $d = 10\%$). The x-axis is log scaled.

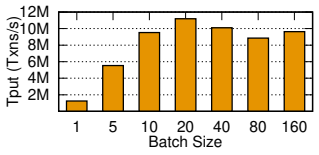


Figure 15: Peak timestamp server throughput under different batch sizes.

results show that batching with fine-grained re-execution is crucial in reaping the benefits of caching.

Peak Performance of the Timestamp Server. We measured the timestamp server's throughput using a simulated FDB-micro workload, where database nodes send requests to the timestamp server without executing transactions or commit logic. Figure 15 shows that the timestamp server's peak throughput is 11.2M Txns/s when the batch size is 20. Peak throughput reduces as batch sizes increase because larger batches are more likely to contend with each other.

6 Related Work

Transactions with co-located computation and storage. Beyond COCO [40] and Sundial [63] discussed in § 5.1.1, several other systems execute transaction on storage nodes [15–17, 30, 40, 43, 49]. Spanner, CockroackDB, and MySQL cluster use two-phase locking during transaction execution followed by 2PC. Granola [17], H-Store [28, 30], and Rococo [41] leverage transactions' one-shot property to execute independent stored procedure fragments at remote partitions. Compared to Hackwrench, these systems commit transactions one at a time and have to wait for remote reads. By contrast, Hackwrench uses caching, speculatively executes transactions without communication, and then commits them in a batch. Speculative 2PC [29] lets transactions read distributed transactions' uncommitted data without waiting for the global commit decision. A distributed transaction's abort causes all dependent transactions to abort. Hackwrench avoids the cascading aborts using lightweight dependency tracking and fine-grained re-execution.

Transactions with separate computation and storage nodes. Beyond FoundationDB, which we discussed in § 5.1.1, several other systems use separate machines for transaction execution and data storage [4, 19, 65, 66]. Some designs [6–9, 11, 57] use a distributed shared-log as the storage layer. Sinfonia [4] supports mini-transactions with client-side caching and uses a modified OCC protocol. However, it aborts invalid transactions due to cache staleness or conflicts. AWS Aurora [53, 54] extends the storage nodes to support log processing, allowing database nodes to broadcast redo logs to storage nodes for parallel processing. Both single-master Aurora and Deuteronomy [33] cache data at the single database node and batch transaction commit. Single-master PolarDB [12] leverages PolarFS as the storage layer but does not cache data. Multi-master Aurora [32] supports read-write transaction processing at multiple database nodes, while little has been published about its design, existing documentation advises users not to simultaneously modify the same tuple from different database nodes. Hackwrench does not impose such a limitation.

Mitigating aborts in distributed databases. Many systems aim to reduce or eliminate conflict-induced aborts in distributed transactions. Granola [17] uses timestamps obtained from loosely synchronized clocks to serialize transactions and eliminate aborts for one-shot transactions with no inter-fragment dependencies. Deterministic databases [20, 21, 35–37, 39, 49, 50] eliminate aborts by ordering transactions before execution, but require tightly coupling compute and storage because the storage layer is responsible for enforcing transaction order which impacts execution logic. Rococo [41] and Janus [42] avoid aborts by deferring execution until the serialization order is determined. Callas [59] and Tebaldi [46] partition transactions into groups, allowing different concurrency controls (e.g., 2PL or OCC) to be used across groups to reduce aborts. Callas [59] also introduces runtime pipelining for the high-contention workload. However, runtime pipelining requires remote synchronization for each transaction, which would limit the benefits of caching and batching. ACC [48] and CormCC [47] partition data and select appropriate concurrency controls for each partition. Finally, other works have suggested co-locating hot data [64] or prioritizing distributed transactions [27], and these approaches are orthogonal to Hackwrench's techniques.

Mitigating aborts in single-machine databases. Transaction healing [58] and Transaction Repair [18] use repair to reduce OCC abort cost in single-node databases. Adapting the idea of repair to the distributed setting is challenging, and to the best of our knowledge, Hackwrench is the first to apply repair in a distributed setting. Many recent proposals also aim to reduce abort rates or blocking time in single-machine databases. IC3 [56] proposes a static analysis-based approach to eliminate unnecessary aborts; Plor [14] uses pessimistic locking and optimistic reads to reduce tail latency for contended workloads; Bamboo [27] reduces the blocking time by "retiring" row locks after the last writes and allowing dirty reads, at the price of additional costs for dependency tracking and cascading aborts; and MOCC [55] avoids OCC validation failures by selectively using read locks for reads likely to cause read-write conflicts. Hackwrench can adapt these protocols to coordinate the local execution at database nodes.

7 Conclusion

We describe Hackwrench, a design for distributed databases which uses best-effort caching and batched validation/commit to execute transactions at multiple database nodes with remote storage. Hackwrench uses fine-grained repair to mitigate the harmful effect of increased transaction invalidation due to stale cache reads and batched validation. Our evaluations with TPC-C show that Hackwrench can achieve much higher throughput than an OCC implementation, FoundationDB, Calvin, COCO, and Sundial.

References

[1] Abadi, D. J., and Faleiro, J. M. An overview of deterministic database systems. CACM, 61(9), September 2018.

[2] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2016).

[3] Adya, A. Weak consistency: A generalized theory and optimistic implementations for distributed transactions. *Ph.D. Thesis* (1999).

[4] AGUILERA, M., MERCHANT, A., SHAH, M., VEITCH, A., AND KARAMANOLIS, C. Sinfonia: a new paradigm for building scalable distributed systems. In *SOSP* (2007).

[5] APPLE. Foundationdb. "https://github.com/apple/foundationdb", 2023.

[6] BALAKRISHNAN, M., FLINN, J., SHEN, C., DHARAMSHI, M., JAFRI, A., SHI, X., GHOSH, S., HASSAN, H., SAGAR, A., SHI, R., LIU, J., GRUSZCZYNSKI, F., ZHANG, X., HOANG, H., YOSSEF, A., RICHARD, F., AND SONG, Y. J. Virtual consensus in delos. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020* (2020), USENIX Association, pp. 617–632.

[7] BALAKRISHNAN, M., MALKHI, D., PRABHAKARAN, V., WOBBLER, T., WEI, M., AND DAVIS, J. D. Corfu: A shared log design for flash clusters. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)* (2012).

[8] BALAKRISHNAN, M., MALKHI, D., WOBBER, T., WU, M., PRABHAKARAN, V., WEI, M., DAVIS, J. D., RAO, S., ZOU, T., AND ZUCKK, A. Tango: Distributed data structures over a shared log. In *SOSP* (2013).

[9] BALAKRISHNAN, M., SHEN, C., JAFRI, A., MAPARA, S., GERAGHTY, D., FLINN, J., VENKAT, V., NEDELCHEV, I., GHOSH, S., DHARAMSHI, M., LIU, J., GRUSZCZYNSKI, F., LI, J., TIBREWAL, R., ZAVERI, A., NAGAR, R., YOSSEF, A., RICHARD, F., AND SONG, Y. J. Log-structured protocols in delos. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021* (2021), R. van Renesse and N. Zeldovich, Eds., ACM, pp. 538–552.

[10] BERNSTEIN, P. A., AND GOODMAN, N. Multiversion concurrency control—theory and algorithms. *ACM Trans. Database Syst. 8*, 4 (dec 1983), 465–483.

[11] BERNSTEIN, P. A., REID, C. W., AND DAS, S. Hyder: A transactional record manager for shared flash. In *Conference on Innovative Data Systems Research (CIDR)* (2011).

[12] CAO, W., LIU, Z., WANG, P., CHEN, S., ZHU, C., ZHENG, S., WANG, Y., AND MA, G. Polarfs: An ultra-low latency and failure resilient distributed file system for shared storage cloud database. *Proc. VLDB Endow. 11*, 12 (2018), 1849–1862.

[13] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2006).

[14] CHEN, Y., YU, X., KOUTRIS, P., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., AND SHU, J. Plor: General transactions with predictable, low tail latency. In *Proceedings of the 2022 International Conference on Management of Data* (New York, NY, USA, 2022), SIGMOD '22, Association for Computing Machinery, p. 19–33.

[15] Cockroachdb design document. "https://github.com/cockroachdb/cockroach/blob/master/docs/design.md", 2021.

[16] CORBETT, J., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., HSIEH, W., KANTHAK, S., KOGAN, E., LI, H., LLOYD, A., MELNIK, S., MWAURA, D., NAGLE, D., QUINLAN, S., RAO, R., ROLIG, L., SAITO, Y., SZYMANIAK, M., TAYLOR, C., WANG, R., AND WOODFORD, D. Spanner: Google's globally-distributed database. In *OSDI* (2012).

[17] COWLING, J., AND LISKOV, B. Granola: low-overhead distributed transaction coordination. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)* (2012), pp. 223–235.

[18] DASHTI, M., JOHN, S. B., SHAIKHHA, A., AND KOCH, C. Transaction repair for multi-version concurrency control. In *SIGMOD '17* (2017).

[19] DEWITT, D., AND GRAY, J. Parallel database systems: the future of high performance database systems. In *Communications and the ACM* (1992).

[20] DONG, Z., TANG, C., WANG, J., WANG, Z., CHEN, H., AND ZANG, B. Optimistic transaction processing in deterministic database. *J. Comput. Sci. Technol. 35*, 2 (2020), 382–394.

[21] FALEIRO, J. M., THOMSON, A., AND ABADI, D. J. Lazy evaluation of transactions in database systems. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2014), SIGMOD '14, Association for Computing Machinery, p. 15–26.

[22] GHARACHORLOO, K., LENOSKI, D., LAUDON, J., GIBBONS, P., GUPTA, A., AND HENNESSY, J. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *ISCA* (1990).

[23] GRAY, J., AND REUTER, A. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1992.

[24] GRAY, J., SUNDARESAN, P., ENGLERT, S., BACLAWSKI, K., AND WEINBERGER, P. J. Quickly generating billion-record synthetic databases. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, USA, May 24-27, 1994* (1994), R. T. Snodgrass and M. Winslett, Eds., ACM Press, pp. 243–252.

[25] GRAY, J. N. Notes on Data Base Operating Systems. *Operating systems* (1978), 393–481.

[26] GRAY, J. N., LORIE, R. A., AND PUTZOLU, G. R. Granularity of locks in a shared data base. In *VLDB* (1975).

[27] GUO, Z., WU, K., YAN, C., AND YU, X. Releasing locks as early as you can: Reducing contention of hotspots by violating two-phase locking. In *Proceedings of the 2021 International Conference on Management of Data* (New York, NY, USA, 2021), SIGMOD '21, Association for Computing Machinery, p. 658–670.

[28] JONES, E. P., ABADI, D. J., AND MADDEN, S. Low overhead concurrency control for partitioned main memory databases. In *SIGMOD '10: Proceedings of the 2010 international conference on Management of data* (2010).

[29] JONES, E. P. C., ABADI, D. J., AND MADDEN, S. Low overhead concurrency control for partitioned main memory databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010* (2010), A. K. Elmagarmid and D. Agrawal, Eds., ACM, pp. 603–614.

[30] KALLMAN, R., KIMURA, H., NATKINS, J., PAVLO, A., RASIN, A., ZDONIK, S., JONES, E. P. C., MADDEN, S., STONEBRAKER, M., ZHANG, Y., HUGG, J., AND ABADI, D. J. H-store: a high-performance, distributed main memory transaction processing system. In *VLDB* (2008).

[31] KUNG, H. T., AND ROBINSON, J. On optimistic methods for concurrency control. In *ACM Transactions on Database Systems (TODS)* (1981).

[32] LEVANDOSKI, J. Aurora Multimaster. HPTS, 2019.

[33] LEVANDOSKI, J., LOMET, D., SENGUPTA, S., STUTSMAN, R., AND WANG, R. High performance transactions in deuteronomy. In *Conference on Innovative Data Systems Research (CIDR)* (2015).

[34] LI, K. Ivy: A shared virtual memory system for parallel computing. In *International Conference on Parallel Processing* (1988).

[35] LIN, Y., PI, S., LIAO, M., TSAI, C., ELMORE, A. J., AND WU, S. Mgcrab: Transaction crabbing for live migration in deterministic database systems. *Proc. VLDB Endow. 12*, 5 (2019), 597–610.

[36] LIN, Y., TSAI, C., LIN, T., CHANG, Y., AND WU, S. Don't look back, look into the future: Prescient data partitioning and migration for deterministic database systems. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021* (2021), G. Li, Z. Li, S. Idreos, and D. Srivastava, Eds., ACM, pp. 1156–1168.

[37] LIU, Y., SU, L., SHAH, V., ZHOU, Y., AND VAZ SALLES, M. A. Hybrid deterministic and nondeterministic execution of transactions in actor systems. In *Proceedings of the 2022 International Conference on Management of Data* (New York, NY, USA, 2022), SIGMOD '22, Association for Computing Machinery, p. 65–78.

[38] LU, Y. coco. "https://github.com/luyi0619/coco", 2023.

[39] LU, Y., YU, X., CAO, L., AND MADDEN, S. Aria: A fast and practical deterministic OLTP database. *Proc. VLDB Endow. 13*, 11 (2020), 2047–2060.

[40] LU, Y., YU, X., CAO, L., AND MADDEN, S. Epoch-based commit and replication in distributed OLTP databases. *Proc. VLDB Endow. 14*, 5 (2021), 743–756.

[41] MU, S., CUI, Y., ZHANG, Y., LLOYD, W., AND LI, J. Extracting more concurrency from distributed transactions. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (2014), pp. 479–494.

[42] MU, S., NELSON, L., LLOYD, W., AND LI, J. Consolidating concurrency control and consensus for commits under conflicts. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (2016), pp. 517–532.

[43] Mysql cluster 8.0. "https://www.mysql.com/products/cluster", 2021.

[44] SOVRAN, Y., POWER, R., AGUILERA, M. K., AND LI, J. Transactional storage for geo-replicated systems. In *ACM Symposium on Operating Systems Principles (SOSP)* (2011).

[45] STONEBRAKER, M., MADDEN, S., ABADI, D. J., HARIZOPOULOS, S., HACHEM, N., AND HELLAND, P. The end of an architectural era (it's time for a complete rewrite). In *VLDB* (2007).

[46] SU, C., CROOKS, N., DING, C., ALVISI, L., AND XIE, C. Bringing modular concurrency control to the next level. In *Proceedings of the 2017 ACM International Conference on Management of Data* (2017), pp. 283–297.

[47] TANG, D., AND ELMORE, A. J. Toward coordination-free and reconfigurable mixed concurrency control. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (USA, 2018), USENIX ATC '18, USENIX Association, p. 809–822.

[48] TANG, D., JIANG, H., AND ELMORE, A. J. 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).

[49] THOMSON, A., DIAMOND, T., WENG, S.-C., REN, K., SHAO, P., AND ABADI, D. J. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (2012), pp. 1–12.

[50] THOMSON, A., DIAMOND, T., WENG, S.-C., REN, K., SHAO, P., AND ABADI, D. J. Fast distributed transactions and strongly consistent replication for oltp database systems. *ACM Trans. Database Syst. 39*, 2 (may 2014).

[51] Tpc-c. "http://www.tpc.org/tpcc/".

[52] TU, S., ZHENG, W., KOHLER, E., LISKOV, B., AND MADDEN, S. Speedy transactions in multicore in-memory databases. In *SOSP* (2013).

[53] VERBITSKI, A., GUPTA, A., SAHA, D., BRAHMADESAM, M., GUPTA, K. K., MITTAL, R., KRISHNAMURTHY, S., MAURICE, S., KHARATISHVILI, T., AND BAO, X. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *SIGMOD '17* (2017).

[54] VERBITSKI, A., GUPTA, A., SAHA, D., COREY, J., GUPTA, K. K., BRAHMADESAM, M., MITTAL, R., KRISHNAMURTHY, S., MAURICE, S., KHARATISHVILI,

T., AND BAO, X. Amazon aurora: On avoiding distributed consensus for i/os, commits, and membership changes. In *SIGMOD '18* (2018).

[55] WANG, T., AND KIMURA, H. Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores. *Proc. VLDB Endow. 10*, 2 (oct 2016), 49–60.

[56] WANG, Z., MU, S., CUI, Y., YI, H., CHEN, H., AND LI, J. Scaling multicore databases via constrained parallel execution. In *Proceedings of the 2016 International Conference on Management of Data* (2016), pp. 1643–1658.

[57] WEI, M., TAI, A., ROSSBACH, C. J., ABRAHAM, I., MUNSHED, M., DHAWAN, M., STABILE, J., WIEDER, U., FRITCHIE, S., SWANSON, S., FREEDMAN, M. J., AND MALKHI, D. vcorfu: A cloud-scale object store on a shared log. In *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017* (2017), A. Akella and J. Howell, Eds., USENIX Association, pp. 35–49.

[58] WU, Y., CHAN, C. Y., AND TAN, K.-L. Transaction healing: Scaling optimistic concurrency control on multicores. In *SIGMOD '16* (2016).

[59] XIE, C., SU, C., LITTLEY, C., ALVISI, L., KAPRITSOS, M., AND WANG, Y. High-performance acid via modular concurrency control. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), pp. 279–294.

[60] YALEDB. Calvin. "https://github.com/yaledb/calvin", 2023.

[61] YANG, L., YAN, X., AND WONG, B. Natto: Providing distributed transaction prioritization for high-contention workloads. In *Proceedings of the 2022 International Conference on Management of Data* (New York, NY, USA, 2022), SIGMOD '22, Association for Computing Machinery, p. 715–729.

[62] YU, X. Sundial. "https://github.com/yxymit/Sundial", 2023.

[63] YU, X., XIA, Y., PAVLO, A., SANCHEZ, D., RUDOLPH, L., AND DEVADAS, S. Sundial: Harmonizing concurrency control and caching in a distributed oltp database management system. In *PVLDB* (2018).

[64] ZAMANIAN, E., SHUN, J., BINNIG, C., AND KRASKA, T. Chiller: Contention-centric transaction execution and data partitioning for modern networks. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020* (2020), ACM, pp. 511–526.

[65] ZHOU, J., XU, M., SHRAER, A., NAMASIVAYAM, B., MILLER, A., TSCHANNEN, E., ATHERTON, S., BEAMON, A. J., SEARS, R., LEACH, J., ET AL. Foundationdb: A distributed unbundled transactional key value store. In *Proceedings of the 2021 International Conference on Management of Data* (2021), pp. 2653–2666.

[66] ZHU, T., ZHAO, Z., LI, F., QIAN, W., ZHOU, A., XIE, D., STUTSMAN, R., LI, H., AND HU, H. Solar: Towards a Shared-Everything database on distributed Log-Structured storage. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (Boston, MA, July 2018), USENIX Association, pp. 795–807.