Fine-Grained Re-Execution for Efficient Batched Commit of Distributed Transactions

Zhiyuan Dong
Zhaoguo Wang
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
zydong829@sjtu.edu.cn
zhaoguowang@sjtu.edu.cn

Xiaodong Zhang
Xian Xu
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
xdzhang97@sjtu.edu.cn
jason73@sjtu.edu.cn

Changgeng Zhao
Department of Computer Science, New York University
changgeng@cs.nyu.edu

Haibo Chen
Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University
haibochen@sjtu.edu.cn

Aurojit Panda
Department of Computer Science, New York University
apanda@cs.nyu.edu

Jinyang Li
Department of Computer Science, New York University
jinyang@cs.nyu.edu

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.

PVLDB Reference Format:
Zhiyuan Dong, Zhaoguo Wang, Xiaodong Zhang, Xian Xu, Changgeng Zhao, Haibo Chen, Aurojit Panda, and Jinyang Li. Fine-Grained Re-Execution for Efficient Batched Commit of Distributed Transactions. PVLDB, 16(8): 1930 - 1943, 2023.
doi:10.14778/3594512.3594523

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/SJTU-IPADS/hackwrench.

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 [14], CockroachDB [13], H-Store [29], and FoundationDB [64]. 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 remotely (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 [14] or OCC-style [3] validation, followed by two-phase commit (2PC) [3, 14, 39]. 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 for realizing our goal. Caching data 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. The cloud database Aurora [52, 53] 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.

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 [3] performs best-effort caching. However, under contention, cached reads can miss writes recently committed by

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.

Proceedings of the VLDB Endowment, Vol. 16, No. 8 ISSN 2150-8097.
doi:10.14778/3594512.3594523

other nodes, causing the corresponding transactions to abort. When transactions are batched together for validation, the invalidation of a transaction will cause other transactions in the batch to abort. Thus, COCO [39] 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 OLTP transaction system. Hackwrench performs best-effort caching and batched commit 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 aborting an entire transaction batch. Transactions are fixed by re-executing operations affected by stale and invalidated reads.

To support fine-grained re-execution, Hackwrench transactions are expressed as a dataflow graph of operations to make their dependencies explicit. Hackwrench uses 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 [25]) to execute and commit transactions locally. Transactions can read uncommitted data of any locally-committed transaction without waiting for its global commit, so as not to be blocked. The resulting dependencies are tracked by the database node and 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. Additionally, Hackwrench's global commit introduces two variations. First, upon validation failure, the database node repairs 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 request. Second, Hackwrench relies on a timestamp server to determine the commit order of batches. Storage nodes validate batches in the order of their commit timestamps, ensuring that repair only needs to happen at most once.

For a common but restricted class of transactions called one-shot transactions [29, 44], we optimize the global commit by offloading the repair to storage nodes. One-shot transactions' dataflow graphes 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 request. This optimization allows storage nodes to handle prepare requests without blocking, avoiding two-phase commit costs.

We have implemented Hackwrench as a distributed transaction system and compared it with a baseline OCC system, Calvin [49], FoundationDB [64], COCO [39], and Sundial [62]. 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 experiments show that Hackwrench's fine-grained repair mechanism greatly reduces the abort overhead when commits are batched. To summarize, the paper makes the following contributions:

- We introduce a new distributed OLTP database system design, Hackwrench, which exploits caching and batching to reduce communication during transaction execution and commit. In particular, we propose a tiered commit protocol to validate and commit 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 and batched commit, we propose fine-grained re-execution to "fix" stale or invalidated reads instead of doing traditional wholesale abort-and-retry, greatly lowering the cost of transaction conflicts.
- For one-shot transactions [29, 44], 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.
- We build a prototype of Hackwrench, which outperforms existing commercial and research systems including FoundationDB [64], COCO [39], Calvin [49], and Sundial [62] in comparable settings.

2 BACKGROUND AND MOTIVATION

2.1 The Cost of Distributed OLTP Transactions

Distributed OLTP transactions incur heavy costs because of their intrinsic need for cross-node 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 nodes separate from storage nodes (e.g., Spanner [14], FoundationDB [64], CockroachDB [13], and MySql NDB Cluster [42]), all reads are remote. For "co-located" systems, aka systems that execute transactions on worker threads co-located with storage nodes (e.g. Calvin [49], H-Store [29], COCO [39], and Sundial [62]), a fraction of the reads in a "multi-partition" transactions[1] must contact some remote nodes.

The second is remote synchronization, needed for ensuring serializability, which can take several forms: i) distributed two-phase locking [25], e.g., used by Spanner [14]); ii) OCC validation [30], e.g., used by COCO [39]; iii) two-phase commit (2PC) [22, 23], which ensures that a committed transaction's data is durable on all relevant nodes. To reduce round-trips, many systems also merge OCC validation with the first phase of 2PC, e.g., Sinfornia [3], Granola [15].

Compared to local execution, remote communication drastically increases transaction latency and decreases system throughput. Throughput is 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 target a "remote-storage" architecture where database nodes that execute transactions are separate from storage nodes that store data. During transaction execution, database nodes read from storage nodes and buffer writes locally. To commit a transaction, the database

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

Table 1: The performance impact of caching and batching. p is the percentage of multi-warehouse NewOrder transaction. “Naive OCC” is similar to COCO [39], 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.

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

node must first validate its reads with the relevant storage nodes. Below, we discuss how the two common optimizations, caching and batching, can be applied 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. When only a single database node can process write transactions (e.g. Deuteronomy [32], single-master Aurora [52]), 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 [3]. Thus, caching increases transaction aborts, which may explain why most systems choose not to cache [13, 14, 42, 64].

Batching. To amortize the cost of remote synchronization, database nodes can batch a group of transactions after execution to validate and commit them together at the storage nodes. Prior works have proposed batched transaction commits, but not validation. For example, single-writer Aurora [52, 53] only batches the writes of transactions for replication to storage nodes. COCO [39] validates transactions individually and then batches validated transactions for replication. In our setting, database nodes must validate a transaction's reads at storage nodes, involving cross-node synchronization. Therefore, it is imperative to batch the validation of transactions. Many tough design questions arise. Should we permit a 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, using 18 Amazon EC2 m5.2xlarge instances (6 database nodes and 12 storage nodes). We increase the multi-warehouse NewOrder transaction possibility (p) to control cross-node conflicts. According to Table 1, caching increases throughput by 58.4% ~ 69.8%. Batching further improves performance by 219.5% ($p = 0\%$) and 144.2% ($p = 9.6\%$). However, the improvement drops to 30.3% ($p = 89.3\%$), as i) batched validation increases the likelihood of conflicts between batches, and ii) a single aborted transaction causes the entire batch of transactions to abort. Basic batching prohibits transactions from reading uncommitted data of those batches still being validated by remote storage nodes. We also included a variation (+RU) that allows reading from such uncommitted batches. Under 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 challenge of sustaining the high performance of “+RU” in the face of low to moderate contention.

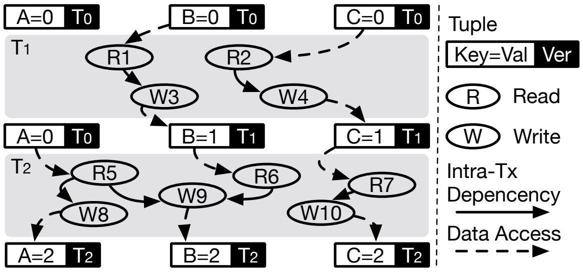


Figure 1: Fine-grained tracking of operation dependencies within a batch of transactions, T_1 and T_2.

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 serializability within a node. In the second stage, referred to as “global commit”, each node groups a batch of locally committed transactions and communicates with storage nodes to validate/persist the batch's read/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 can batch together dependent transactions. Otherwise, we would be restricted to only batching together transactions with non-overlapping data accesses, which seriously constrains 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 transactions must be aborted if they observed any aborted writes. To mitigate the abort cost, we need a more efficient solution than aborting and retrying a batch of transactions. Our insight is that it is cheaper to repair the transactions batch by selectively re-executing operations affected by stale or invalid reads.

We implement 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

[2] Traditional cache consistency protocols [21, 33] 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

dependencies among a batch of two locally committed transactions that access three data tuples A, B, and C. We assume a tuple's version 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, 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 storage nodes. 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 [16, 57], but has several differences. First, since repairing is done in a batch of transactions, 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 [16] because the latter lacks precision.

3 HACKWRENCH DESIGN

System overview. Figure 2 depicts Hackwrench's architecture, which has three 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. We use the term *logical* storage node to refer to a group of *physical* storage nodes that replicate the same data partition (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 [52]. The size of the read-/write quorum is configurable and must ensure a non-empty quorum intersection. Hackwrench's tiered commit protocol guarantees strict serializability. It uses the timestamp server to ensure a consistent ordering of global commits from different database nodes.

Hackwrench relies on a Paxos-replicated configuration service to maintain the system configuration (aka view), similar to other distributed storage systems [11, 39, 43]. The view includes the identity of the timestamp server and the mapping of each data partition to its logical storage node. Each view is identified with a monotonically increasing view number. RPC requests contain the highest view number known to the sender. The timestamp server and storage nodes reject requests whose view numbers do not match theirs.

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.

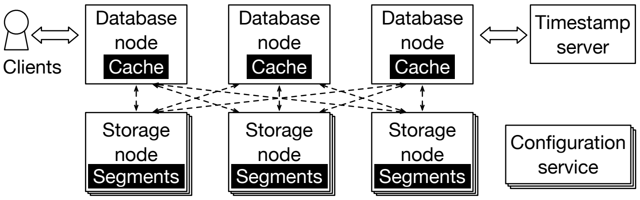


Figure 2: The architecture of Hackwrench. Dashed lines indicate communication links and stacked boxes indicate replicas.

Each database node keeps a large in-memory tuple-level cache. 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 [44]. Unlike other systems [39, 41, 49, 51, 55, 58] 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, whose outputs only depend on the 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. We store one copy of the static dataflow graph for a given transaction type at each node.

In Hackwrench's tiered commit protocol, the concurrency control 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 (§ 3.3).

Execution and local commit. To execute transactions, a database node reads from its data cache whenever possible and buffers writes locally. It uses two-phase locking [25] (with NO_WAIT for deadlock prevention) to ensure strict serializability. 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. To ensure that transactions are enqueued in the local commit order, the database node releases the locks held by 2PL after the transaction are pushed into the queue.

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, the downside is: if transaction T aborts 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.

3.3 Global Commit

Hackwrench dequeues a batch of locally committed transactions and tries to globally commit them at 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 [3], the *prepare* phase validates the batch's read set and persists its write set (in the form of redo log) for crash recovery. We introduce two crucial variations. First, Hackwrench relies on a centralized timestamp server to assign the batch of transactions a consistent commit ordering. Second, Hackwrench repairs conflicted transactions instead of aborting the whole transaction batch. Next, we describe the commit timestamp assignment and the global commit procedure 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 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 strategy is to impose a total order on commit timestamps, which is straightforward and adopted by many systems, e.g. FoundationDB [64], Spanner [14], and Deterministic DB [1, 49]. Instead, we use partial ordering for commit timestamps, which guarantees the consistent ordering of conflicting transactions while avoiding unnecessary blocking of non-conflicting batches.

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, 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 commit timestamps' partial order. 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 writer batch of 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. While commit timestamps are segment-level, storage nodes do tuple-level locking and validation to minimize false blocking and false conflicts. Thanks to coarse-grained timestamps and batched timestamp assignment, 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 deduplication. The batch's read and write set determine the set of participating logical storage nodes involved in the global commit.

After assembling a batch, the database node fetches a commit timestamp from the timestamp server (Line 3). The timestamp server handles the same database node's requests in order, ensuring that commit timestamps are consistent with batches' local commit order.

When receiving the commit timestamp, the database node proceeds to the *prepare* phase. It merges the batch's read set and write set. Then it sends Prepare requests in parallel to all participating logical storage nodes (Lines 6,7). The Prepare requests contain the batch's commit timestamp, read set, write set, and transactions' inputs.

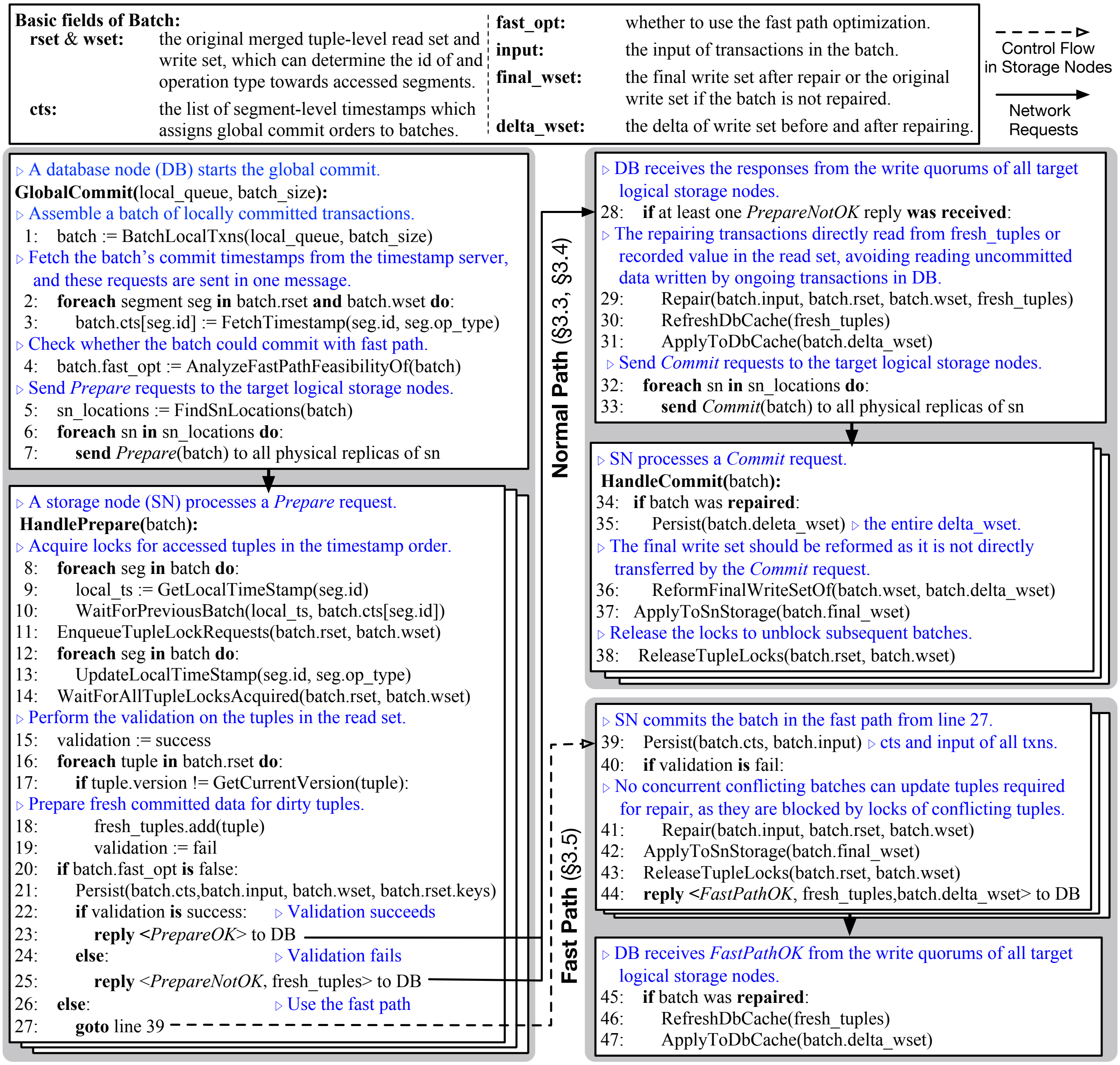
Upon receiving a batch's Prepare request, the storage node acquires tuple-level locks for the batch, following the commit timestamp's partial order. It first checks whether this request must wait for other batches to finish enqueuing their lock requests (Line 10), by comparing its responsible segments' 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 conflicts between 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



Basic fields of Batch:

rset & wset: the original merged tuple-level read set and write set, which can determine the id of and operation type towards accessed segments.

cts: the list of segment-level timestamps which assigns global commit orders to batches.

fast_opt: whether to use the fast path optimization.

input: the input of transactions in the batch.

final_wset: the final write set after repair or the original write set if the batch is not repaired.

delta_wset: the delta of write set before and after repairing.

```
▷ A database node (DB) starts the global commit.
GlobalCommit(local_queue, batch_size):
▷ Assemble a batch of locally committed transactions.
 1:  batch := BatchLocalTxns(local_queue, batch_size)
▷ Fetch the batch's commit timestamps from the timestamp server,
  and these requests are sent in one message.
 2:  foreach segment seg in batch.rset and batch.wset do:
 3:      batch.cts[seg.id] := FetchTimestamp(seg.id, seg.op_type)
▷ Check whether the batch could commit with fast path.
 4:  batch.fast_opt := AnalyzeFastPathFeasibilityOf(batch)
▷ Send Prepare requests to the target logical storage nodes.
 5:  sn_locations := FindSnLocations(batch)
 6:  foreach sn in sn_locations do:
 7:      send Prepare(batch) to all physical replicas of sn
```

```
▷ A storage node (SN) processes a Prepare request.
HandlePrepare(batch):
▷ Acquire locks for accessed tuples in the timestamp order.
 8:  foreach seg in batch do:
 9:      local_ts := GetLocalTimeStamp(seg.id)
10:      WaitForPreviousBatch(local_ts, batch.cts[seg.id])
11:  EnqueueTupleLockRequests(batch.rset, batch.wset)
12:  foreach seg in batch do:
13:      UpdateLocalTimeStamp(seg.id, seg.op_type)
14:  WaitForAllTupleLocksAcquired(batch.rset, batch.wset)
▷ Perform the validation on the tuples in the read set.
15:  validation := success
16:  foreach tuple in batch.rset do:
17:      if tuple.version != GetCurrentVersion(tuple):
▷ Prepare fresh committed data for dirty tuples.
18:          fresh_tuples.add(tuple)
19:          validation := fail
20:  if batch.fast_opt is false:
21:      Persist(batch.cts,batch.input, batch.wset, batch.rset.keys)
22:      if validation is success:        ▷ Validation succeeds
23:          reply <PrepareOK> to DB
24:      else:                            ▷ Validation fails
25:          reply <PrepareNotOK, fresh_tuples> to DB
26:  else:                                ▷ Use the fast path
27:      goto line 39
```

```
▷ DB receives the responses from the write quorums of all target
  logical storage nodes.
28:  if at least one PrepareNotOK reply was received:
▷ The repairing transactions directly read from fresh_tuples or
  recorded value in the read set, avoiding reading uncommitted
  data written by ongoing transactions in DB.
29:      Repair(batch.input, batch.rset, batch.wset, fresh_tuples)
30:      RefreshDbCache(fresh_tuples)
31:      ApplyToDbCache(batch.delta_wset)
 ▷ Send Commit requests to the target logical storage nodes.
32:  foreach sn in sn_locations do:
33:      send Commit(batch) to all physical replicas of sn
```

```
▷ SN processes a Commit request.
HandleCommit(batch):
34:  if batch was repaired:
35:      Persist(batch.deleta_wset) ▷ the entire delta_wset.
▷ The final write set should be reformed as it is not directly
  transferred by the Commit request.
36:      ReformFinalWriteSetOf(batch.wset, batch.delta_wset)
37:  ApplyToSnStorage(batch.final_wset)
▷ Release the locks to unblock subsequent batches.
38:  ReleaseTupleLocks(batch.rset, batch.wset)
```

```
▷ SN commits the batch in the fast path from line 27.
39:  Persist(batch.cts, batch.input) ▷ cts and input of all txns.
40:  if validation is fail:
▷ No concurrent conflicting batches can update tuples required
  for repair, as they are blocked by locks of conflicting tuples.
41:      Repair(batch.input, batch.rset, batch.wset)
42:  ApplyToSnStorage(batch.final_wset)
43:  ReleaseTupleLocks(batch.rset, batch.wset)
44:  reply <FastPathOK, fresh_tuples,batch.delta_wset> to DB
```

```
▷ DB receives FastPathOK from the write quorums of all target
  logical storage nodes.
45:  if batch was repaired:
46:      RefreshDbCache(fresh_tuples)
47:      ApplyToDbCache(batch.delta_wset)
```

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

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 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).

When repair finishes, the database node sends Commit requests to all participating storage nodes (Lines 32, 33). Instead of transferring the final write set ($final\_wset$), it 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 locks (Lines 37,38).

Our design guarantees the success of repair except for two scenarios. The first scenario is when a read or write operation changes its key during re-execution. For repair to succeed, tuple-level locks are 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 [18, 20, 34, 35, 49, 50, 60].

3.5 Fast-Path Optimization

A common but restricted form of transaction is the so-called one-shot transaction [29, 44]. 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 repair and commit T_1 and T_2.

Figure 3's bottom right corner shows fast path's pseudocode. In this case, every logical storage node persists the commit timestamp and input of *all* transactions in the batch (Line 39), which are 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 the write set (Lines 46-47).

3.6 Failure Recovery and Correctness

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 handling from database failures, we must recover all pending transactions from the failed database node. This is because unfinished transactions can block conflicting transactions from other database nodes. Our design uses a replicated configuration service to detect failures and initiate recovery. Upon detecting a database node failure, the configuration service notifies all storage nodes to 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 then decides whether each pending batch should be committed (because sufficient information is available to do so), or aborted, and we describe both cases below. Recovery ends 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 request, 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 Prepare requests (containing the commit timestamp, transaction inputs, and read set and write set keys) to recompute the final write set. We re-execute the transaction using the tuples stored at the storage nodes, the timestamp, and the transaction input. Similar to the repair logic, our recovery logic aborts transactions whose keys in the original read or write set change during re-execution.

On the other hand, if the batch was committed at a logical storage node, we must make the same decision. However, the transaction cannot be simply re-executed. 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 (instead of the relevant part) before committing the batch. During recovery, the coordinator reads this delta write set from a logical storage node that committed the transaction, and combines it with the write set from the persisted Prepare request to construct the final write set and commit the batch.

Transactions processed with the fast-path optimization are handled slightly differently because storage nodes independently make commit decisions. Our recovery procedure depends on two design choices: first, Prepare requests 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 in a batch that uses the fast path. 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. Handling these failures requires global quiescence and view changes. To start the view change, the configuration service informs all storage nodes to stop processing new batches and finish all pending ones, using the scheme for recovering database nodes. Once finished, the configuration service is safe to broadcast a new view (containing the new timestamp server). The new timestamp server assigns timestamps starting from the initial value. Storage nodes only process batches with timestamps of the current view. View change can be used to add or remove storage nodes but we do not implement it currently.

Correctness. Due to space constraints, we leave the proof of correctness of Hackwrench's design in an extended version of the paper [19].

4 IMPLEMENTATION

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 combining all pending transactions into a batch, which has two drawbacks: 1) it limits parallelism because non-conflicting transactions execute sequentially, and 2) repairs to a transaction can unnecessarily delay a non-conflicting transaction's commit. Therefore, Hackwrench splits a batch into non-conflicting sub-batches for 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 which we merge into a sub-batch. Different sub-batches in a batch access different segments, and thus can be validated, repaired, and committed independently.

Dependency tracking. Hackwrench tracks dependencies among transaction operations (the dashed lines in Figure 1). For any read Op_r, it stores the <*key*,*version*> pairs of accessed tuples. 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 re-execution is constructing a large dependency graph for all transactions in one batch and repairing accordingly. However, this approach introduces runtime overhead for dependency graph construction and traversal. Hackwrench uses a simple but efficient way instead. As transactions within the same batch are totally ordered (§ 3.3), 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 their dependent reads 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 six systems. Among them, Hackwrench, an OCC system, and FoundationDB [64] use remote storage. COCO [39], Calvin [49], and Sundial [62] use co-located storage. All systems store data in memory. Three-way replication is enabled except for Sundial. We used the original implementation from GitHub [4, 37, 59, 61] for all systems.

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. It uses 1) two-phase locking [25] (with NO_WAIT) to handle local conflicts, 2) optimistic concurrency control [30] 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 in Table 1. However, caching, batching, and uncommitted reads have similar impacts on the performance.

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

Calvin. Calvin is a deterministic database that assigns total order to batched transactions before execution and avoids 2PC overhead. Each node has a single thread scheduling transaction execution, which limits parallelism. Unlike Calvin, Hackwrench assigns commit order after transaction execution and 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.

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 locks remote tuples for write operations, 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. The Sundial implementation does not support replication.

COCO and Sundial's design assumes that storage and transaction processing are co-located. Consequently, using their protocols with remote storage introduces message overheads due to validating up-to-date tuples stored in remote storage. These systems, unlike

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

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

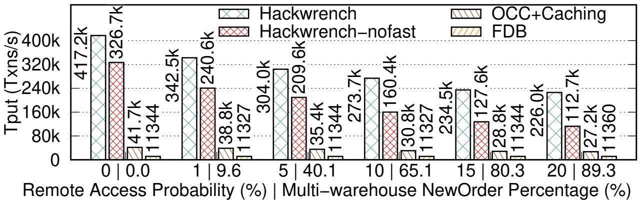


Figure 4: Comparison of systems with remote storage on TPC-C. The multi-warehouse NewOrder transaction percentage is calculated from the remote access probability.

Hackwrench, don't implement tiered commits and thus incur the message overhead even when the transaction is aborted due to local intra-node conflicts. We measure these overheads using modified implementations of both (COCO-remote and Sundial-remote in the graphs) that use remote storage. Our modification forces any single-node transaction[3] in COCO or Sundial to do a 2PC procedure with one remote node. 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 node. To distinguish, We refer to COCO-colocated and Sundial-colocated as COCO and Sundial's original implementation, respectively.

5.1.2 Benchmarks and workloads. We use two benchmarks for our evaluation: (a) FoundationDB's **FDB-Micro** microbenchmark [64]; and (b) the **TPC-C** benchmark [48].

FDB-Micro. This benchmark has 214M tuples with 8-byte key and a 24-byte value. The tuples are partitioned evenly across storage nodes and database nodes, respectively. 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 only accesses tuples in the local database node's 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. We run the standard TPC-C benchmark with five transaction types. In the benchmark standard, 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 route transactions to the responsible database nodes. 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 support batching and reading uncommitted data. Except for FDB, all systems' performance drops as the remote access probability increases. It is because FDB does not use caching (which reduces the chances of stale reads) and has a relatively low throughput.

Latency. Table 2 shows the zero-load latencies 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. FDB has the highest latency because it does not cache data. Although OCC also does not use caching, it reads multiple tuples in one round trip and thus has lower latency than FDB.

5.3 Factor analysis

Design breakdown. Figure 5 uses TPC-C to measure the impacts of Hackwrench's design decisions. We use OCC as the baseline. Caching improves performance by 76.7% ~ 80.5%. Batching and uncommitted reads ("+RU") improve performance by 7.1× for the $r = 0\%$ case, but

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

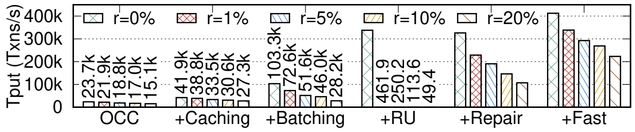


Figure 5: Design breakdown on TPC-C. The designs are applied incrementally based on OCC. "+Caching": adding local caches. "+Batching": supporting batching. "+RU": permitting uncommitted reads across batches. "+Repair": using repair. "+Fast": enabling fast-path optimization.

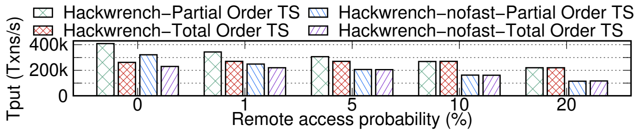


Figure 6: Partial order and total order timestamps comparison.

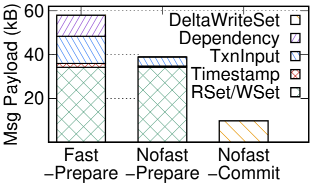


Figure 7: Message payloads.

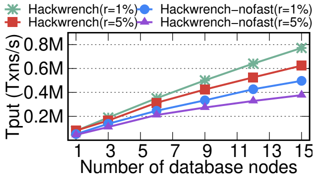


Figure 8: Scalability (TPC-C).

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 [64]. Partial ordering improves performance significantly when remote access probabilities are low. This is because a total-order design requires database nodes to broadcast any transaction batch to all storage nodes for progress, on the other hand with partial ordering we only need to send transaction batches to relevant storage nodes, reducing overheads.

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

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 storage nodes 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) is sent in the Prepare requests.

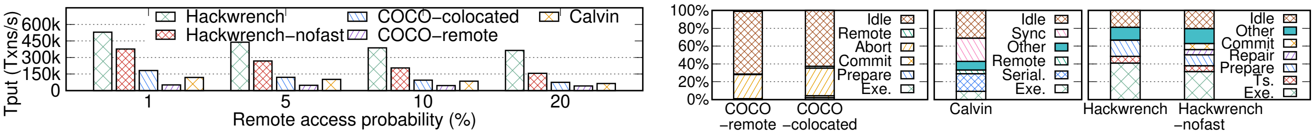
5.4 Scalability

Figure 8 shows Hackwrench's and Hackwrench-nofast's scalability on standard TPC-C. 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. We found that the timestamp server was not a scalability bottleneck due to the use of segment-level timestamps and batching (§ 5.6).

5.5 Comparison to Co-located Systems

As COCO's implementation only supports TPC-C NewOrder and Payment transactions [39], we use a mixture of 50% NewOrder and 50% Payment transactions to compare against COCO and Calvin. All systems use 19 nodes and used 48 warehouses. Figure 9(a) shows that Hackwrench and Hackwrench-nofast outperform the other systems. We provide a breakdown of throughput and CPU time in Table 3 and Figure 9(b). 'Thread' refers to the thread for transaction execution. Hackwrench and Hackwrench-nofast have higher per-thread throughput than both COCO variants. COCO-remote and COCO-colocated spent only 0.36% and 1.95% of CPU time executing transactions. Their CPUs remain idle for the majority of the time. It is because COCO does not batch validation and uses only one ongoing transaction per thread, and COCO's threads sleep 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 Hackwrench to avoid idling. Since Calvin executes each transaction on all node replicas, it has 7 ($\lceil 19/3 \rceil = 7$) (replicated) database nodes in total. It uses 9.17% CPU time for execution, while 19.9% and 26.7% CPU time are spent by data serialization and synchronization in its TPC-C benchmark implementation. The ratio of execution time between different systems does not necessarily correspond to the ratio of per-thread throughput, as transaction execution latency can vary across systems. Hackwrench's throughput is 9.27× COCO-remote's, 3.9× COCO-colocated's, and 5.7× Calvin's.

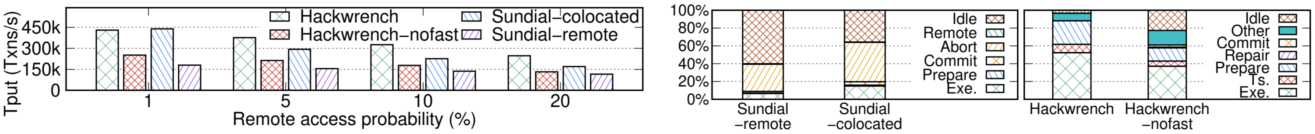
Our comparison with Sundial uses the standard TPC-C but disables replication. All systems use 11 nodes and 48 warehouses. Figure 10(a) shows that Sundial-colocated has the highest throughput for $r = 1\%$ as 77.3% transactions are single-node and benefit from the co-located architecture. However, because Sundial does



(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.

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

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 4: Throughput breakdown for Figure 10(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

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-node. Relatively, Hackwrench-nofast outperforms Sundial-remote for all r. Table 4 and Figure 10(b) show detailed throughput and CPU breakdowns. Like COCO, each Sundial thread allows only one ongoing transaction and sleeps before retrying, wasting CPU time. Only 7.0% and 15.1% CPU times of Sundial-remote and Sundial-colocated are used for transaction execution, while those of Hackwrench and Hackwrench-nofast are 52.4% and 37.9%.

5.6 FDB-Micro

Impact of Contention and Distributed Transactions. According to Figure 11, under low contention, Hackwrench and Hackwrench-nofast have similar performance. COCO-colocated performs best for $d \leq 20\%$ due to a large fraction of single-node transactions. As d increases, its performance drops and eventually becomes similar to COCO-remote's. Under high contention, COCO-colocated and COCO-remote have a performance gap of 56.6% for d = 100%, as COCO-remote involves network overhead for resolving local conflict. Hackwrench and Hackwrench-nofast's performance gap also increases as more batches are repaired (4.4% ~ 27.8% v.s. 53.4% ~ 99.7%).

Note that our FDB-Micro implementation for Calvin does not impose noticeable serialization and synchronization overheads. The bottleneck is the single-threaded scheduler under low contention. In the high-contention setting, Calvin achieves 92.5% ~ 98.7% of its low-contention throughput and has performance comparable to Hackwrench-nofast, because it orders transactions before execution, avoiding aborts. OCC+Caching performs badly due to frequent aborts caused by its use of local 2PL with NO_WAIT, and policy of reading committed data. Lastly, FDB outperforms OCC+Caching due to the use of more database nodes (12 vs 6) and its implementation of FDB-Micro, which retrieves all tuples in a single network round trip.

In Figure 12, Hackwrench's and Hackwrench-nofast's performance is higher than that in Figure 11 because disabling replication saves CPU time for replication and the number of database nodes remains unchanged. Sundial's trends are similar to those of COCO.

Impact of Batching. Figure 13 varies batch size from 1 to 160. Increasing batch sizes allow Hackwrench to support a higher offered load 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×. In the high-contention case, the peak throughput improves by 81.7%, 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 under high contention. On average, one remote conflict causes 2.23 to 441.4 transactions to repair with increasing batch sizes.

Impact of Caching. Figure 14 controls the cache miss rate c for each data access. As c decreases, Hackwrench's performance improves by up to 2.1×. 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 results show that batching with fine-grained re-execution is crucial in reaping the benefits of caching.

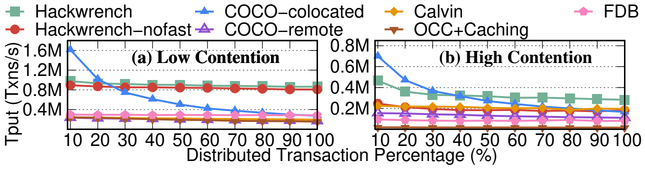


Figure 11: FDB-micro performance with replication (27 nodes).

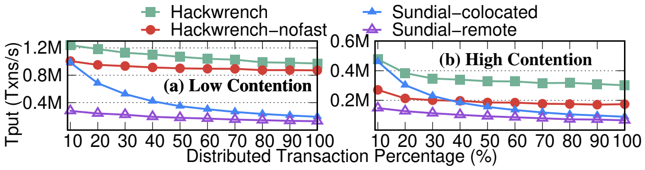


Figure 12: FDB-micro performance without replication (15 nodes).

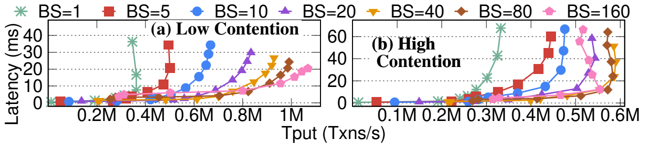


Figure 13: Impact of batching on FDB-micro (low contention, $d = 10\%$). "BS" stands for batch size.

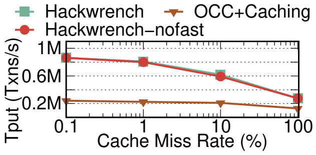


Figure 14: Impact of caching on FDB-micro (low contention, d = 10%). The x-axis is log scaled.

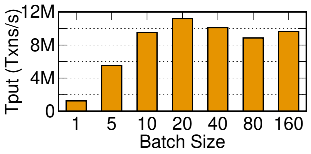


Figure 15: Peak timestamp server throughput under different batch sizes.

The Peak Performance of Timestamp Server. We conducted a simulated FDB-micro workload, where database nodes send requests to the timestamp server without transaction execution and commit. Figure 15 shows that the 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 [39] and Sundial [62] discussed in § 5.1.1, several other systems use co-located storage [13–15, 29, 39, 42, 49]. Spanner, CockroachDB, and MySQL Cluster use 2PL and 2PC for transaction execution and commit. Granola [15], H-Store [27, 29], and Rococo [40] leverage transactions' one-shot property to execute independent stored procedure fragments at remote partitions. Compared to Hackwrench, they do not support caching and batching. Speculative 2PC [28] allows reading distributed transactions' uncommitted data without waiting for the commit. It suffers from cascading abort overheads, which can be reduced by Hackwrench's repair mechanism.

Transactions with separate computation and storage nodes. Beyond FoundationDB discussed in § 5.1.1, several other systems use remote storage [3, 17, 64, 65]. Some designs [5–9, 56] use a distributed shared-log as the storage layer. Sinfonia [3] 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 [52, 53] 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 [32] cache data at the single database node and batch transaction commit. Single-master PolarDB [10] leverages PolarFS as the storage layer but does not cache data. Multi-master Aurora [31] processes read-write transaction at multiple database nodes. Although its design is not published, existing documentation suggests not simultaneously modifying 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 [15] uses timestamps obtained from loosely synchronized clocks to serialize transactions and reduce aborts. Deterministic databases [18, 20, 34–36, 38, 49, 50] eliminate aborts by ordering transactions before execution. Rococo [40] and Janus [41] avoid aborts by deferring execution until the serialization order is determined. Callas [58] and Tebaldi [45] partition transactions into groups, allowing different concurrency controls (e.g., 2PL or OCC) to be used across groups to reduce aborts. Callas [58] also introduces runtime pipelining for the high-contention workload, which requires remote synchronization for each transaction, limiting the benefits of caching and batching. ACC [47] and CormCC [46] partition data and select appropriate concurrency controls for each partition. Finally, other works have suggested co-locating hot data [63] or prioritizing distributed transactions [26], and they are orthogonal to Hackwrench.

Mitigating aborts in single-machine databases. Transaction Healing [57] and Transaction Repair [16] use repair to reduce single-node databases' abort cost. IC3 [55] leverages static analysis to eliminate aborts; Plor [12] uses pessimistic locking and optimistic reads to reduce tail latency for contended workloads; Bamboo [26] reduces blocking time by "retiring" row locks after the last writes and allowing dirty reads; and MOCC [54] avoids OCC validation failures by using read locks for reads likely to cause conflicts. Hackwrench can adapt these protocols to coordinate database nodes' local execution.

7 CONCLUSION

We describe Hackwrench, a design for distributed databases using best-effort caching and batched commit for transactions execution at multiple database nodes with remote storage. Hackwrench uses fine-grained repair to mitigate the harmful effect of increased transaction conflicts due to stale cache reads and batched validation. TPC-C results show that Hackwrench achieves much higher throughput than an OCC system, FoundationDB, Calvin, COCO, and Sundial.

ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China under Grant Nos. 62272304 and 62132014. Jinyang Li is partially supported by NSF 2220407. Aurojit Panda is partially supported by NSF 2145471. Zhaoguo Wang is the corresponding author.

REFERENCES

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

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 265–283.

[3] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. 2007. Sinfonia: A New Paradigm for Building Scalable Distributed Systems. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles* (Stevenson, Washington, USA) *(SOSP '07)*. Association for Computing Machinery, New York, NY, USA, 159–174. https://doi.org/10.1145/1294261.1294278

[4] Apple. 2023. Foundationdb. "https://github.com/apple/foundationdb". Last accessed April 2023.

[5] Mahesh Balakrishnan, Jason Flinn, Chen Shen, Mihir Dharamshi, Ahmed Jafri, Xiao Shi, Santosh Ghosh, Hazem Hassan, Aaryaman Sagar, Rhed Shi, Jingming Liu, Filip Gruszczynski, Xianan Zhang, Huy Hoang, Ahmed Yossef, Francois Richard, and Yee Jiun Song. 2020. Virtual Consensus in Delos. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, USA, Article 35, 16 pages.

[6] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobbler, Michael Wei, and John D. Davis. 2012. CORFU: A Shared Log Design for Flash Clusters. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX Association, San Jose, CA, 1–14. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/balakrishnan

[7] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, Tao Zou, and Aviad Zuck. 2013. Tango: Distributed Data Structures over a Shared Log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 325–340. https://doi.org/10.1145/2517349.2522732

[8] Mahesh Balakrishnan, Chen Shen, Ahmed Jafri, Suyog Mapara, David Geraghty, Jason Flinn, Vidhya Venkat, Ivailo Nedelchev, Santosh Ghosh, Mihir Dharamshi, Jingming Liu, Filip Gruszczynski, Jun Li, Rounak Tibrewal, Ali Zaveri, Rajeev Nagar, Ahmed Yossef, Francois Richard, and Yee Jiun Song. 2021. Log-Structured Protocols in Delos. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 538–552. https://doi.org/10.1145/3477132.3483544

[9] Philip A. Bernstein, Colin W. Reid, and Sudipto Das. 2011. Hyder - A Transactional Record Manager for Shared Flash. In *Fifth Biennial Conference on Innovative Data Systems Research, CIDR 2011, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*. www.cidrdb.org, Asilomar, CA, USA, 9–20. http://cidrdb.org/cidr2011/Papers/CIDR11_Paper2.pdf

[10] Wei Cao, Zhenjun Liu, Peng Wang, Sen Chen, Caifeng Zhu, Song Zheng, Yuhui Wang, and Guoqing Ma. 2018. PolarFS: An Ultra-low Latency and Failure Resilient Distributed File System for Shared Storage Cloud Database. *Proc. VLDB Endow.* 11, 12 (2018), 1849–1862. https://doi.org/10.14778/3229863.3229872

[11] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2006. Bigtable: A Distributed Storage System for Structured Data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7* (Seattle, WA) *(OSDI '06)*. USENIX Association, USA, 15.

[12] Youmin Chen, Xiangyao Yu, Paraschos Koutris, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Jiwu Shu. 2022. Plor: General Transactions with Predictable, Low Tail Latency. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 19–33. https://doi.org/10.1145/3514221.3517879

[13] cockroachdb. 2021. CockroachDB Design Document. "https://github.com/cockroachdb/cockroach/blob/master/docs/design.md". Last accessed April 2023.

[14] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2013. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.* 31, 3, Article 8 (aug 2013), 22 pages. https://doi.org/10.1145/2491245

[15] James Cowling and Barbara Liskov. 2012. Granola: Low-Overhead Distributed Transaction Coordination. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. USENIX Association, Boston, MA, 223–235. https://www.usenix.org/conference/atc12/technical-sessions/presentation/cowling

[16] Mohammad Dashti, Sachin Basil John, Amir Shaikhha, and Christoph Koch. 2017. Transaction Repair for Multi-Version Concurrency Control. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 235–250. https://doi.org/10.1145/3035918.3035919

[17] David DeWitt and Jim Gray. 1992. Parallel Database Systems: The Future of High Performance Database Systems. *Commun. ACM* 35, 6 (jun 1992), 85–98. https://doi.org/10.1145/129888.129894

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

[19] Zhiyuan Dong, Zhaoguo Wang, Xiaodonng Zhang, Xian Xu, Changgeng Zhao, Haibo Chen, Aurojit Panda, and Jinyang Li. 2023. Fine-Grained Re-Execution for Efficient Batched Commit of Distributed Transactions (Extended Version). "https://ipads.se.sjtu.edu.cn/_media/publications/hackwrench.pdf". Last accessed April 2023.

[20] Jose M. Faleiro, Alexander Thomson, and Daniel J. Abadi. 2014. Lazy Evaluation of Transactions in Database Systems. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 15–26. https://doi.org/10.1145/2588555.2610529

[21] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. 1990. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture* (Seattle, Washington, USA) *(ISCA '90)*. Association for Computing Machinery, New York, NY, USA, 15–26. https://doi.org/10.1145/325164.325102

[22] Jim Gray. 1978. Notes on Data Base Operating Systems. In *Operating Systems, An Advanced Course*. Springer-Verlag, Berlin, Heidelberg, 393–481.

[23] Jim Gray and Andreas Reuter. 1992. *Transaction Processing: Concepts and Techniques* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[24] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J. Weinberger. 1994. Quickly Generating Billion-Record Synthetic Databases. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data* (Minneapolis, Minnesota, USA) *(SIGMOD '94)*. Association for Computing Machinery, New York, NY, USA, 243–252. https://doi.org/10.1145/191839.191886

[25] J. N. Gray, R. A. Lorie, and G. R. Putzolu. 1975. Granularity of Locks in a Shared Data Base. In *Proceedings of the 1st International Conference on Very Large Data Bases* (Framingham, Massachusetts) *(VLDB '75)*. Association for Computing Machinery, New York, NY, USA, 428–451. https://doi.org/10.1145/1282480.1282513

[26] Zhihan Guo, Kan Wu, Cong Yan, and Xiangyao Yu. 2021. Releasing Locks As Early As You Can: Reducing Contention of Hotspots by Violating Two-Phase Locking. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 658–670. https://doi.org/10.1145/3448016.3457294

[27] Evan P.C. Jones, Daniel J. Abadi, and Samuel Madden. 2010. Low Overhead Concurrency Control for Partitioned Main Memory Databases. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (Indianapolis, Indiana, USA) *(SIGMOD '10)*. Association for Computing Machinery, New York, NY, USA, 603–614. https://doi.org/10.1145/1807167.1807233

[28] Evan P.C. Jones, Daniel J. Abadi, and Samuel Madden. 2010. Low Overhead Concurrency Control for Partitioned Main Memory Databases. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (Indianapolis, Indiana, USA) *(SIGMOD '10)*. Association for Computing Machinery, New York, NY, USA, 603–614. https://doi.org/10.1145/1807167.1807233

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

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

[31] Justin Levandoski. 2019. Aurora Multimaster. "http://www.hpts.ws/papers/2019/aurora-multimaster-hpts2019.pdf". Last accessed April 2023.

[32] Justin J. Levandoski, David B. Lomet, Sudipta Sengupta, Ryan Stutsman, and Rui Wang. 2015. High Performance Transactions in Deuteronomy. In *Seventh Biennial Conference on Innovative Data Systems Research, CIDR 2015, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*. www.cidrdb.org, Asilomar, CA, USA. http://cidrdb.org/cidr2015/Papers/CIDR15_Paper15.pdf

[33] Kai Li. 1988. IVY: A Shared Virtual Memory System for Parallel Computing. In *Proceedings of the International Conference on Parallel Processing, ICPP '88, The Pennsylvania State University, University Park, PA, USA, August 1988. Volume 2: Software*. Pennsylvania State University Press, University Park, PA, USA, 94–101.

[34] Yu-Shan Lin, Shao-Kan Pi, Meng-Kai Liao, Ching Tsai, Aaron J. Elmore, and Shan-Hung Wu. 2019. MgCrab: Transaction Crabbing for Live Migration in Deterministic Database Systems. *Proc. VLDB Endow.* 12, 5 (2019), 597–610. https://doi.org/10.14778/3303753.3303764

[35] Yu-Shan Lin, Ching Tsai, Tz-Yu Lin, Yun-Sheng Chang, and Shan-Hung Wu. 2021. Don't Look Back, Look into the Future: Prescient Data Partitioning and Migration for Deterministic Database Systems. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 1156–1168. https://doi.org/10.1145/3448016.3452827

[36] Yijian Liu, Li Su, Vivek Shah, Yongluan Zhou, and Marcos Antonio Vaz Salles. 2022. Hybrid Deterministic and Nondeterministic Execution of Transactions in Actor Systems. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 65–78. https://doi.org/10.1145/3514221.3526172

[37] Yi Lu. 2023. coco. "https://github.com/luyi0619/coco". Last accessed April 2023.

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

[39] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2021. Epoch-based Commit and Replication in Distributed OLTP Databases. *Proc. VLDB Endow.* 14, 5 (2021), 743–756. https://doi.org/10.14778/3446095.3446098

[40] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. 2014. Extracting More Concurrency from Distributed Transactions. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 479–494. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/mu

[41] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. 2016. Consolidating Concurrency Control and Consensus for Commits under Conflicts. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 517–532. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/mu

[42] Oracle. 2023. MySQL Cluster 8.0. "https://www.mysql.com/products/cluster". Last accessed April 2023.

[43] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. 2011. Transactional Storage for Geo-Replicated Systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal) *(SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 385–400. https://doi.org/10.1145/2043556.2043592

[44] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. 2007. The End of an Architectural Era: (It's Time for a Complete Rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB '07)*. VLDB Endowment, Vienna, Austria, 1150–1160.

[45] Chunzhi Su, Natacha Crooks, Cong Ding, Lorenzo Alvisi, and Chao Xie. 2017. Bringing Modular Concurrency Control to the Next Level. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 283–297. https://doi.org/10.1145/3035918.3064031

[46] Dixin Tang and Aaron J. Elmore. 2018. Toward Coordination-free and Reconfigurable Mixed Concurrency Control. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 809–822. https://www.usenix.org/conference/atc18/presentation/tang

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

[48] The Transaction Processing Council. 2022. TPC benchmark C. "http://www.tpc.org/tpcc/". Last accessed April 2023.

[49] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/2213836.2213838

[50] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2014. Fast Distributed Transactions and Strongly Consistent Replication for OLTP Database Systems. *ACM Trans. Database Syst.* 39, 2, Article 11 (may 2014), 39 pages. https://doi.org/10.1145/2556685

[51] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-Memory Databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 18–32. https://doi.org/10.1145/2517349.2522713

[52] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1041–1052. https://doi.org/10.1145/3035918.3056101

[53] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, James Corey, Kamal Gupta, Murali Brahmadesam, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvilli, and Xiaofeng Bao. 2018. Amazon Aurora: On Avoiding Distributed Consensus for I/Os, Commits, and Membership Changes. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 789–796. https://doi.org/10.1145/3183713.3196937

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

[55] Zhaoguo Wang, Shuai Mu, Yang Cui, Han Yi, Haibo Chen, and Jinyang Li. 2016. Scaling Multicore Databases via Constrained Parallel Execution. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1643–1658. https://doi.org/10.1145/2882903.2882934

[56] Michael Wei, Amy Tai, Christopher J. Rossbach, Ittai Abraham, Maithem Munshed, Medhavi Dhawan, Jim Stabile, Udi Wieder, Scott Fritchie, Steven Swanson, Michael J. Freedman, and Dahlia Malkhi. 2017. vCorfu: A Cloud-Scale Object Store on a Shared Log. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 35–49. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/wei-michael

[57] Yingjun Wu, Chee-Yong Chan, and Kian-Lee Tan. 2016. Transaction Healing: Scaling Optimistic Concurrency Control on Multicores. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1689–1704. https://doi.org/10.1145/2882903.2915202

[58] Chao Xie, Chunzhi Su, Cody Littley, Lorenzo Alvisi, Manos Kapritsos, and Yang Wang. 2015. High-Performance ACID via Modular Concurrency Control. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 279–294. https://doi.org/10.1145/2815400.2815430

[59] Yaledb. 2023. Calvin. "https://github.com/yaledb/calvin". Last accessed April 2023.

[60] Linguan Yang, Xinan Yan, and Bernard Wong. 2022. Natto: Providing Distributed Transaction Prioritization for High-Contention Workloads. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 715–729. https://doi.org/10.1145/3514221.3526161

[61] Xiangyao Yu. 2023. Sundial. "https://github.com/yxymit/Sundial". Last accessed April 2023.

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

[63] Erfan Zamanian, Julian Shun, Carsten Binnig, and Tim Kraska. 2020. Chiller: Contention-Centric Transaction Execution and Data Partitioning for Modern Networks. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 511–526. https://doi.org/10.1145/3318464.3389724

[64] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, Evan Tschannen, Steve Atherton, Andrew J. Beamon, Rusty Sears, John Leach, Dave Rosenthal, Xin Dong, Will Wilson, Ben Collins, David Scherer, Alec Grieser, Young Liu, Alvin Moore, Bhaskar Muppana, Xiaoge Su, and Vishesh Yadav. 2021. FoundationDB: A Distributed Unbundled Transactional Key Value Store. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 2653–2666. https://doi.org/10.1145/3448016.3457559

[65] Tao Zhu, Zhuoyue Zhao, Feifei Li, Weining Qian, Aoying Zhou, Dong Xie, Ryan Stutsman, Haining Li, and Huiqi Hu. 2018. Solar: Towards a Shared-Everything Database on Distributed Log-Structured Storage. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 795–807. https://www.usenix.org/conference/atc18/presentation/zhu