Accelerating Skewed Workloads With Performance Multipliers in the TurboDB Distributed Database

Jennifer Lam, Jeffrey Helt, and Wyatt Lloyd, *Princeton University;*
Haonan Lu, *University at Buffalo*

https://www.usenix.org/conference/nsdi24/presentation/lam

This paper is included in the
Proceedings of the 21st USENIX Symposium on
Networked Systems Design and Implementation.

April 16–18, 2024 • Santa Clara, CA, USA

978-1-939133-39-7

Open access to the Proceedings of the
21st USENIX Symposium on Networked
Systems Design and Implementation
is sponsored by

Accelerating Skewed Workloads With Performance Multipliers in the TurboDB Distributed Database

Jennifer Lam*, Jeffrey Helt*, Wyatt Lloyd*, Haonan Lu†
*Princeton University, †University at Buffalo

Abstract

Distributed databases suffer from performance degradation under skewed workloads. Such workloads cause high contention, which is exacerbated by cross-node network latencies. In contrast, single-machine databases better handle skewed workloads because their centralized nature enables performance optimizations that execute contended requests more efficiently. Based on this insight, we propose a novel hybrid architecture that employs a single-machine database inside a distributed database and present TurboDB, the first distributed database that leverages this hybrid architecture to achieve up to an order of magnitude better performance than representative solutions under skewed workloads.

TurboDB introduces two designs to tackle the core challenges unique to its hybrid architecture. First, Hybrid Concurrency Control is a specialized technique that coordinates the single-machine and distributed databases to collectively ensure process-ordered serializability. Second, Phalanx Replication provides fault tolerance for the single-machine database without significantly sacrificing its performance benefits. We implement TurboDB using CockroachDB and Cicada as the distributed and single-machine databases, respectively. Our evaluation shows that TurboDB significantly improves the performance of CockroachDB under skewed workloads.

1 Introduction

Distributed databases support large applications by sharding data across many machines to provide capacity far greater than what can fit on a single machine. However, these databases often experience severe performance degradation under *skewed workloads* where most requests contend on a small subset of data. This contention results in excessive aborts and retries that are expensive in the distributed setting. Unfortunately, many real-world workloads are highly skewed [3, 9, 10, 62].

In contrast, single-machine databases store all data on one machine. Although these databases cannot support large-scale applications, they handle skewed workloads more efficiently. For instance, Cicada [37], a single-machine database, can achieve much higher throughput and lower latency for TPC-C New-Order than CockroachDB, a representative distributed database, running on 48 servers [54]. This drastic difference stems from two *performance multipliers*, such as local concurrency control and one-stop execution, which single-machine databases can employ due to their centralized nature, but distributed databases cannot.

Local concurrency control techniques handle conflicts more efficiently by leveraging global knowledge of, and centralized control over, transactions, as all transactions access the same machine. However, when data is spread over multiple servers, these techniques, such as memory fences in Silo [58] and shared lock tables in MVTL [1], are infeasible.

One-stop execution, which handles transactions entirely within a single machine, enables shorter transaction lifetimes. For instance, a lock's acquisition and release, as part of transaction execution on a single machine, takes only nanoseconds to microseconds. However, distributed lock management, which requires multiple round trips between servers, takes orders of magnitude longer. Short transaction lifetimes lower the likelihood of conflicts and thus aborts. Aborting and retrying distributed transactions is more costly than aborting local transactions, due to network delays.

Empowered by local concurrency control and one-stop execution, single-machine databases offer a natural solution to the challenge of skewed workloads. Therefore, this paper proposes a novel hybrid architecture that employs a single-machine database within a distributed database to improve performance under skewed workloads. The single-machine database "turbocharges" overall performance under skew while the distributed database scales capacity.

Specifically, one server of the distributed database is designated as the *turbo*, which runs a single-machine database. The turbo co-locates many popular, contended data items, creating a focal point on which the single-machine database can concentrate its performance multipliers. The remaining servers run a distributed database to handle less contended requests, which access less popular data.

TurboDB is the first distributed database to employ this hybrid architecture. The architecture enables it to achieve significantly better performance than representative distributed databases under skewed workloads. However, it requires TurboDB to overcome two new correctness challenges.

First, a transaction may access data on both the single-

machine and distributed databases. Thus, some of its requests are executed with local concurrency control employed by the turbo while the rest of its requests are executed with distributed concurrency control. TurboDB must ensure the transaction as a whole is isolated from other transactions, and all transactions provide the correct consistency guarantees. Second, TurboDB must make the turbo fault tolerant. Although there are replication techniques for single-machine databases, none automatically work for TurboDB because they require full control of transactions, which the turbo does not have, i.e., the turbo is part of the distributed database, and its execution partially relies on the rest of the system.

TurboDB addresses these challenges, while preserving the performance benefits of the turbo, through two novel designs: Hybrid Concurrency Control and Phalanx Replication.

Hybrid Concurrency Control (HCC) leverages timestamp ordering to stitch together local and distributed concurrency control protocols. HCC ensures all requests of the same transaction commit at the same timestamp, and that all transactions are serialized in their timestamp order, thus guaranteeing process-ordered serializability [16, 40]. To maximize the performance advantages of local concurrency control, HCC applies *finale commit*, a serial-commit protocol that avoids unnecessary blocking and aborts on the turbo.

Phalanx Replication tackles the unique challenge of replicating a single-machine database in a distributed setting. A later-received transaction may be assigned a smaller timestamp due to clock skew and network asynchrony. Thus, existing replication techniques for single-machine databases that rely on assigned timestamps being monotonically increasing do not automatically work for TurboDB's setting. To address this challenge, Phalanx uses Frontline, a mechanism that determines the correct replication order even when timestamps may be out-of-order. Phalanx also employs a set of techniques to reduce replication costs, including per-core replication and decoupled log replay. The non-turbo servers are replicated with standard techniques for distributed databases, e.g., Raft [43].

We implement TurboDB using Cicada [37] and CockroachDB [54], which are representative single-machine and distributed databases, respectively. We evaluate TurboDB using YCSB+T and TPC-C, with a variety of read-write ratios and levels of skew. TurboDB achieves up to an order of magnitude higher throughput and 50% lower latency for highly-skewed YCSB+T, and up to 1.6× higher throughput and lower latency for highly contended TPC-C than CockroachDB.

In summary, this paper makes the following contributions:

- A novel system architecture that incorporates a single-machine database within a distributed database to "turbocharge" the performance under skewed workloads.
- TurboDB, the first design that leverages this new architecture using HCC and Phalanx to ensure the correctness of the combination while retaining the performance benefits of the turbo.
- An implementation and evaluation that shows TurboDB outperforms a representative distributed database by up to an order of magnitude under skewed workloads.

2 Background and Motivation

This section provides background on distributed databases and then discusses the challenge of skewed workloads.

2.1 Distributed Databases

Front-end *client* machines translate user requests into transactions whose requests are executed on the *servers* that store the data. Databases run concurrency control protocols to ensure that transactions appear to take effect in an order that satisfies specific consistency guarantees. TurboDB provides *process-ordered serializability* [16, 40], which guarantees there exists a total order amongst committed transactions, and the total order respects the order in which clients issue transactions. Process-ordered serializability is stronger than snapshot isolation and plain serializability [44].

Fault tolerance. Distributed databases tolerate server failures by replicating each server onto multiple replicas through consensus protocols such as Raft [43].

2.2 The Challenge of Skewed Workloads

Many real-world workloads are highly skewed [2, 4, 8–10, 28, 62]. For instance, Facebook's TAO reports the most popular data items are queried several orders of magnitude more often than other objects [3, 9], and Twitter's Twemcache reports an even higher skew [62]. Skewed workloads are difficult in general and particularly adversarial to distributed databases.

First, skewed workloads introduce more *conflicts*—i.e., concurrent transactions access overlapping (popular) data items with at least one write—in a distributed setting, because distributed concurrency control must coordinate multiple servers, thus prolonging transaction execution due to network transmission times. The longer the execution, the more likely it is that transactions conflict. Conflicts often result in aborts, which are especially expensive as retrying distributed transactions takes a long time, due to network delays.

Second, the overall performance of the database is limited by its performance on the few popular data items as they are accessed by most requests. As a result, the excessive aborts and prolonged execution, due to distributed concurrency control that often incurs multiple rounds of inter-machine communication, on the popular data have a disproportionately large negative effect on the overall performance.

A natural solution. Single-machine databases can better handle skewed workloads because their local concurrency control executes conflicting transactions more efficiently by

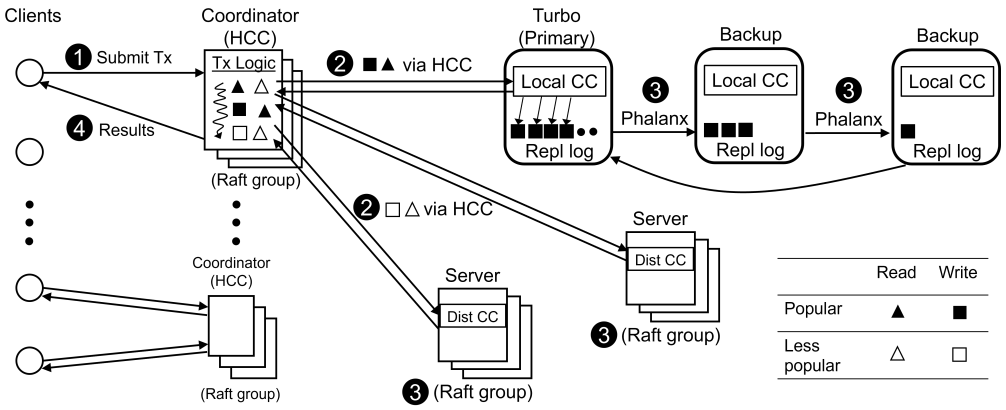


Figure 1: TurboDB designs Hybrid Concurrency Control (HCC) to integrate the turbo and servers. The requests on popular keys are mostly sent to the turbo, which runs local concurrency control and a specialized replication protocol. Less popular requests are processed by servers running distributed concurrency control and a standard replication protocol.

employing techniques that are infeasible in a distributed setting, such as memory fences [58], shared lock tables [1], and single-threaded timestamping [23]. Because no cross-server coordination is needed, transactions can be processed in one stop within the machine, thus greatly shortening the execution time and, in turn, reducing conflicts. These performance multipliers, i.e., local concurrency control and one-stop execution, enable fast processing of popular data items, lifting the performance bottleneck of the overall database.

3 Design Overview

TurboDB is built on a hybrid architecture that incorporates a single-machine database as the turbo. The turbo leverages its performance multipliers to efficiently execute transactions that contend on popular data, enhancing overall performance under skewed workloads.

3.1 A Hybrid Architecture

As shown in Figure 1, TurboDB is built on a standard distributed database and dedicates one of its storage servers to running a single-machine database (the *turbo*). The turbo and remaining *servers* run local and distributed concurrency control protocols, respectively. The turbo can send/receive requests to/from all the servers through RPCs. The servers, as part of the distributed database, are made fault tolerant using standard techniques. The turbo is made fault tolerant through a special technique: Phalanx Replication (§5).

Data placement. To leverage the turbo's performance multipliers for skewed workloads, TurboDB co-locates many popular keys on the turbo. The more popular keys the turbo stores, the greater performance improvement it brings to the system. Given the capacity and load on the turbo, some popular keys may remain on the servers. The information on data locations, i.e., the mapping of a key to the server or turbo that stores it, is stored on each server and can be kept up-to-date via standard techniques, e.g., Zookeeper [66].

3.2 Transaction Life Cycle

As shown in Figure 1, the overall flow of TurboDB executing a transaction with its hybrid architecture is as follows:

❶ A client receives a user request and translates it into a transaction. The client sends the transaction to one of the servers, and this server will serve as the coordinator for processing this transaction.

❷ The coordinator executes the transaction following Hybrid Concurrency Control (§4) by sending its requests to the servers and/or turbo which store the data this transaction accesses.

❸ Committed transactions replicate their state on servers through the standard technique the distributed database uses, e.g., Raft, and replicate their state on the turbo through Phalanx (§5).

❹ The coordinator replies to the client with the results of the transaction after it is committed and replicated, and the client then replies to the user.

Limitations. First, TurboDB assumes that data popularity does not change significantly or abruptly over time, i.e., data popularity changes on relatively slow timescales compared to how fast data can be migrated. To deal with changes in data popularity, TurboDB relies on existing techniques to migrate data between the turbo and servers. Second, co-locating popular keys on the turbo would make it more difficult to react to load spikes, i.e., sudden increases in request rate, and less resilient to failures or slowdowns. TurboDB partially mitigates this issue by not oversubscribing the turbo, i.e., it reserves enough CPUs and memory for moderate load increases.

As the first step in exploring a hybrid database architecture, TurboDB focuses on its core design challenges. We leave investigating the above limitations to future work.

Core challenges. The core challenges in TurboDB's use of a hybrid architecture are ensuring correctness and fault tolerance while preserving the high performance of the turbo.

4 Hybrid Concurrency Control

This section explains Hybrid Concurrency Control (HCC), which orchestrates the local and distributed concurrency control protocols co-existing in TurboDB's architecture. HCC ensures the system as a whole is process-ordered serializable, without significantly sacrificing the turbo's performance.

4.1 HCC Insight

HCC ensures consistency across the whole system by coordinating the local and distributed concurrency control protocols. A naive design would use traditional two-phase locking (with two-phase commit) across the turbo and servers to handle transactions that access both databases. While this ensures that transactions are serialized, it negates the performance benefits of using the turbo. Locking popular keys, even for one RTT across phases, prolongs the transaction's lifetime and reduces concurrency, sacrificing one-stop execution (§2.2).

To maintain the performance multipliers of the turbo, we apply a specialized two-phase protocol (consisting of the execute and commit phases) that does not acquire distributed locks on the turbo in the execute phase and employs *finale commit*, which is a serial two-step mechanism, in the commit phase to reduce unnecessary aborts on the turbo and enable one-stop execution.

To provide process-ordered serializability, HCC leverages timestamp ordering [7] to ensure that both local and distributed concurrency control protocols commit all requests of a transaction at the same timestamp, which represents the transaction's serialization point. HCC thus assumes both local and distributed protocols are timestamp-based, which is true of many existing protocols [14, 37, 41, 54, 58].

4.2 The Execute Phase

The client starts a transaction by generating a unique timestamp, a combination of the client's ID and the current time. Timestamps generated by the same client are strictly increasing. The client sends the transaction and timestamp to a server, which acts as the transaction's coordinator. The timestamp is used to inform the local and distributed protocols to commit all requests of this transaction at this timestamp.

Algorithm 4.1 shows the coordinator logic. In the execute phase, the coordinator buffers writes locally, and issues read requests (lines 3–12). The values returned by these reads may be used to complete any missing key dependencies, e.g., the value returned by a read request determines what to read/write in another request (line 12). The coordinator sends each request and the timestamp to either a server or the turbo, based on the up-to-date key-location mapping it stores (lines 4–8).

Algorithm 4.1: Transaction coordinator logic

```
1  Function HybridConcurrencyControl(tx, t) :
2      results ← {} // transaction results
       // Begin the execute phase
3      for req in tx.read_set do
4          if req.key on turbo then
5              res, is_aborted ← LocalCC(req, t, "read_only")
               // remove turbo reads
6              tx.read_set ← tx.read_set − req
7          else
8              res, is_aborted ← DistributedCC(req, t)
9          if is_aborted is true then
10             exit(tx.abort)
11         results ← results ∪ res
12         UpdateKeyDependencies(tx, res)
       // Begin the commit phase
13     hot_set ← {}
14     for req in tx.write_set do
           // read_set now only has server reads
15         if req.key on turbo then
16             hot_set ← hot_set ∪ req
17             continue
           // Send writes required by DistCC
18         res, is_aborted ← DistributedCC(req, t)
19         if is_aborted is true then
20             exit(tx.abort)
21         results ← results ∪ res
22     for req in hot_set do
23         res, is_aborted ← LocalCC(req, t, "finale_commit")
24         if is_aborted is true then
25             SendAbortMsgToServers(tx)
26             exit(tx.abort)
27         results ← results ∪ res
28     SendCommitMsgToServers(tx)
29     return results
```

Each server executes reads following the distributed concurrency control protocol. The turbo, in contrast, encapsulates these reads as a standalone read-only transaction so that active locks are not left behind after the execute phase, ensuring that these reads do not block other transactions on the turbo. Specifically, the turbo executes the "read-only transaction" with its local concurrency control, by returning the values at the specified timestamp. Each key accessed by these reads stores a piece of metadata, which records that "transaction *tx* read this key at timestamp *t*." This metadata will be used in the commit phase if the same transaction updates the same keys. In such cases, this metadata signals to the turbo that the reads (which were executed as a standalone read-only transaction) in the first phase and the writes in the second phase are from the same transaction, and the turbo will handle the commit accordingly. We discuss the details next.

4.3 The Commit Phase

After all reads are executed and key dependencies are resolved, the transaction enters the commit phase (lines 13–29). A naive design would send the remaining requests, i.e., buffered writes, to the turbo and servers in parallel, committing them with corresponding concurrency control protocols. However, this would cause unnecessary aborts on the turbo, e.g., when the turbo commits its part of the transaction, but the distributed concurrency control aborts the other part, the transaction as a whole must be aborted, wasting the work on the turbo. Moreover, trying to commit in parallel requires extra coordination between the turbo and servers, which would result in prolonged execution on the turbo and forfeit its one-stop execution. Therefore, HCC enforces a serial commit order: it first attempts to commit on the servers before attempting to commit on the turbo.

Finale commit. The coordinator divides the buffered writes into cool and hot sets, which update the servers and the turbo, respectively (lines 13–17). The coordinator sends the cool set and the timestamp to the servers and attempts to commit these writes at the specified timestamp through distributed concurrency control, e.g., two-phase commit (2PC) could be involved. Other messages as part of the distributed concurrency control, e.g., prepare messages for reads, may be sent together, depending on the specific protocol (lines 18–21).

If at least one server decides to abort this transaction, e.g., at the end of the prepare phase of 2PC, the coordinator aborts this transaction without (unnecessarily) attempting to commit the hot set (lines 19 and 20). If the servers agree to commit the transaction, the coordinator sends the hot set and the timestamp to the turbo and attempts to commit these writes at the timestamp through local concurrency control (lines 22–27). If the turbo cannot commit a write because some requests have read the key at this timestamp, the turbo checks if the most recent reads are from the same transaction as this write, e.g., the "read-only transaction" in the execute phase, and allows the write to commit in this case.

If the turbo commits the hot set, the coordinator then sends a commit message to each involved server to finally commit this transaction (line 28); otherwise, the transaction is aborted, and an abort message is sent to each server (line 25). If the transaction commits, the coordinator can respond to the client without waiting for the acknowledgments of the commit messages (line 29), reducing latency by one RTT, a technique used in many systems, e.g., CockroachDB. If the transaction is aborted, it will be retried by the coordinator (lines 24–26).

Finale commit enforces a serial commit order: trying to commit the cool set on servers → servers are ready to commit → trying to commit the hot set on the turbo → the turbo commits → the servers commit. This serial order preserves the turbo's performance multipliers by ensuring that a transaction does not compete for resources on the turbo if it cannot be committed on the servers and that each transaction updates the turbo at most once at the commit time, thus preserving one-stop execution.

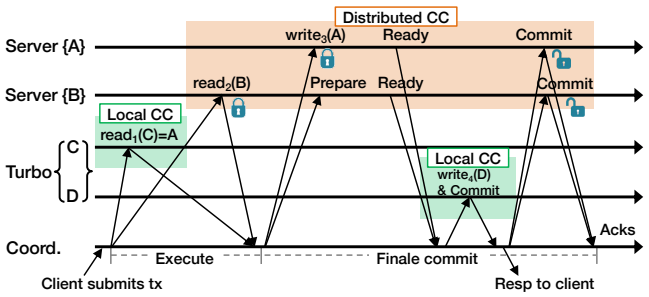


Figure 2: *tx* has 2 reads and 2 writes where the return value of $read_1$ determines which key $write_3$ updates. HCC integrates distributed concurrency control that may hold locks across phases and local concurrency control that does not.

Figure 2 shows an example execution. Transaction *tx* has four requests that include a key dependency: $read_1$ determines which key $write_3$ updates. Keys A and B are on two servers while C and D are stored on the turbo. In the execute phase, $read_1$ is executed by local concurrency control as a standalone read-only transaction without leaving behind locks while $read_2$ holds an active lock following distributed concurrency control (e.g., 2PL). $write_4$ is sent to the turbo in the commit phase only if $read_2$ and $write_3$ are ready to commit on the servers, which will finally commit $read_2$ and $write_3$ if the turbo commits $write_4$.

4.4 Correctness of HCC

This section explains why HCC enforces process-ordered serializability and ensures that transactions eventually terminate.

HCC is safe. HCC guarantees process-ordered serializability by satisfying the following requirements: (1) there exists a total order among all transactions, and (2) the total order respects the process order (§2.1).

First, HCC guarantees (1) by committing all requests of each transaction at the same timestamp, assigned by the client at the transaction's start. Specifically, the pre-assigned timestamp is used by the reads in the execute phase to retrieve values, by the servers to commit the cool set, and by the turbo to commit the hot set. Therefore, the timestamp is the serialization point of each transaction, and the timestamp order is the transactions' commit order. Because timestamps uniquely identify each transaction (§4.2), and timestamp order is a total order, transactions' commit order is total.

Second, HCC guarantees (2). Because each client generates timestamps in a strictly increasing manner (§4.2), later-issued transactions must have larger timestamps than any previously issued transactions by the same client, thus must appear later in the total order, respecting the process order.

HCC is live. Distributed concurrency control guarantees that the requests executed on the servers do not deadlock. Local concurrency control guarantees that the requests executed on the turbo do not deadlock. HCC's finale commit enforces a serial commit order between the servers and turbo, i.e., all transactions follow the same access order: servers $\rightarrow$ the turbo, thus transactions that access both servers and the turbo do not deadlock. Therefore, HCC guarantees that transactions eventually terminate.

5 Phalanx Replication

Another core challenge of TurboDB's hybrid architecture is correctly replicating the turbo for fault tolerance without trading off its performance multipliers.

5.1 Phalanx Insight

A naive solution would be deploying a standard consensus protocol designed for distributed systems, i.e. Raft. However, recent work on single-machine databases [26, 32, 46, 51] has shown that doing so significantly degrades the database's performance. They thus propose special techniques to replicate such databases. Specifically, they assign transactions *strictly increasing* timestamps and ensure that transactions are both committed and replicated in this timestamp order. Leveraging this timestamp order, these techniques remove most of the replication work from the critical path of transaction execution, preserving good performance. While Phalanx leverages the insight of these techniques, it cannot, however, directly apply them to TurboDB's hybrid setting.

The challenge is that these techniques require transactions be timestamped in *strictly increasing* order, which is straightforward for single-machine databases, which have full control over transactions. In contrast, as part of a hybrid database, the turbo passively accepts transactions whose timestamps have been predetermined by the clients. Due to network asynchrony, the turbo may have to execute transactions whose predetermined timestamps are smaller than any it has previously seen. Phalanx must tackle this unique timestamp challenge.

In the following subsections, we first explain the techniques Phalanx uses to preserve the performance of the turbo, then detail the timestamp challenges and Phalanx's timestamp management that tackles the challenges.

5.2 Protocol Basics

Phalanx arranges its replicas in a chain. The head of the chain is the primary; the rest are backups; and the last backup is the tail. The primary is the only replica that communicates with servers and runs local concurrency control, e.g., the "turbo" in Section 4 is this primary replica. Phalanx sequentially propagates the turbo's log, i.e., a sequence of committed updates grouped by transactions, down the chain to each backup. Once the tail receives the log, it sends the primary an acknowledgment (*ack*). Each backup applies the log's updates in an order specified by Phalanx (§5.3).

As a variant of primary-backup [59], Phalanx tolerates f failures with $f+1$ replicas while a coordination service, e.g., ZooKeeper [66], may be used to detect failures and handle membership changes in the replica group.

Decoupled replication. Phalanx preserves the turbo's performance multipliers by decoupling replication from transaction execution, shielding execution from replication delays as much as possible. Specifically, when the primary commits a transaction, it makes its effect immediately visible to future transactions (not yet to users), buffers its response into a response queue, and appends its committed updates to a replication log. The updates in the replication log are asynchronously propagated to the backups, while the primary continues to execute future transactions, i.e., replication does not block transaction execution. The responses of committed transactions are released by the response queue (i.e., they are sent to their coordinators) in order when these transactions' updates have been applied by all backups, i.e., when the primary receives an ack that indicates the completion of replication from the tail.

Allowing the primary to execute future transactions without being blocked by the replication of committed transactions best utilizes the turbo's performance multipliers. It is safe to make committed transactions visible to future transactions before they are replicated, i.e., there is no risk of cascading aborts, because they will certainly commit (the finale-commit guarantees that the turbo being ready to commit the transaction's hot set implies that its cool set must have first been ready to commit on the servers; neither side will abort). Moreover, their results will be visible to the users only after they are successfully replicated.

Per-core replication. Phalanx partitions the replication log across CPU cores. Each core propagates its sub-logs in parallel, reducing inter-core synchronization. Algorithm 5.1 shows the pseudocode. For simplicity, the pseudocode illustrates the primary and backups propagating a single log entry, but our implementation batches entries for performance. When the replication of a transaction starts (line 34), the primary finds an available core, e.g., core 3, and appends a new entry to core 3's sub-log (lines 36-37). This new entry contains this transaction's updates and commit timestamp. Periodically, core 3 propagates new entries to the next replica in the chain (line 41). When the next backup receives these new entries (Algorithm 5.2), it appends them to the sub-log for which its own core 3 is responsible and then propagates these entries to the next replica (lines 46-50). Due to a one-to-one correspondence between cores across all replicas (Figure 3), sub-logs managed by the same core become (eventually) identical across all replicas. The primary's replication log is stored on all backups and partitioned across cores in the same way.

Algorithm 5.1: Phalanx primary handling requests

```
30 Function HANDLEREQUEST(coord, req_msg) :
       // Receive a request msg from a coordinator
31     req ← req_msg.req; t ← req_msg.t; flag_str ← req_msg.flag
32     res, is_aborted ← LOCALCC(req, t, flag_str)
33     if not is_aborted then
34         response_queue.APPEND(coord, res, t)
           // Find a core i to start replication
35         if req is update then
36             log_entry_i ← {req, t}
37             sublog_i.APPEND(log_entry_i)
38             unacked_entries_i ← unacked_entries_i + log_entry_i
               // Update core i's t_s
39             core_i.t_s ← min{core_i.t_s, t}
               // Update the global Frontline if needed
40             t_f ← min{t_f, core_i.t_s}
               // Propagate entry to the next replica
41             tail_ack_i ← next_backup.PROPAGATE(log_entry_i, t_f)
               // Once receiving the ack
42             unacked_entries_i ← unacked_entries_i − log_entry_i
43             core_i.t_s ← min{unacked_entries_i.t}
44             t_f ← min{∀t_s ∈ cores.t_s}
```

Algorithm 5.2: Backup receives propagated entry

```
45 Function PROPAGATE(new_entry, t_f):
       // Append new log entry, then propagate
46     sublog_i.APPEND(new_entry)
47     if this is tail then
           // Tail acks to the primary
48         SENDACK(to:primary, new_entry, t_f)
49     else
           // Propagate down the chain
50         next_backup.PROPAGATE(new_entry, t_f)
       // Apply sublog entries up to t_f (i.e., t < t_f)
51     for entry in sublog_i do
52         if entry.t < t_f then
               // Apply entry to the database
53             this.APPLY(entry)
               // Remove replayed entry from sublog
54             sublog_i ← sublog_i − entry
```

Algorithm 5.3: Primary releases buffered responses

```
55 Function RELEASEBUFFRESPS():
       // Periodically invoked by the primary
56     for resp in response_queue do
57         coord ← resp.coord; res ← resp.res; t ← resp.t
58         if resp.t < t_f then
               // Releases and removes response
59             SENDRESPONSE(to:coord, res, t)
60             response_queue ← response_queue − resp
```

Decoupled log replay. When a backup receives new log entries, it appends these entries to its own sub-log of the corresponding core (line 46), which eventually replays the sub-log, i.e., the core applies the updates in its sub-log to the backup's database (lines 51-54). The backup propagates the new entries to the next backup without waiting for its local replay to complete (line 50). When these new entries finally reach the tail, the tail replays them and sends an *ack* to the primary, indicating that all backups have received these new log entries. Algorithm 5.2 shows the pseudocode.

When the primary receives the *ack* (line 41), the transactions associated with these new entries are considered safe, i.e., their state is stored on all backups and will be replayed by each backup. Yet, Phalanx must ensure a correct log-replay order before the transactions' responses can be returned to their coordinators. That is, Phalanx must guarantee that the transactions take effect on each backup in the same order as the primary, enabling backups to seamlessly and correctly take over the primary's role if the primary fails. Phalanx leverages timestamps to enforce such a log-replay order, and the next section explains a unique challenge Phalanx must tackle and how Phalanx overcomes it.

5.3 Timestamp Challenges & Frontline

Existing solutions execute and replicate transactions in a monotonically increasing timestamp order to ensure that once a transaction finishes, all the transactions before it (e.g., ones whose values this transaction may have read) must have been replicated and returned to their users, since they have smaller timestamps than the current transaction. Yet, the turbo has no control over timestamp generation and thus cannot expect timestamps to be monotonically increasing. Transactions are timestamped by the client machines and may arrive at the turbo in any order. Thus, naively applying existing solutions to TurboDB may lead to two issues.

First, out-of-order timestamps may incur unnecessary stalls, because the system cannot replicate a transaction until it is certain that no future transaction with a smaller timestamp will arrive. That is, later-arriving transactions with smaller timestamps block the replication of earlier transactions that have larger timestamps. For example, in Figure 3, w_3 arrives later than w_1 or w_2 but has a smaller timestamp, then it blocks w_1 and w_2 from replicating. Even worse, the system can never be certain when such transactions (e.g., w_3) may arrive or if they even exist.

Second, we cannot naively disobey the timestamp order, lest incorrect behavior arise during failover, as shown in Figure 3. Let's say w_1 and w_2 belong to tx_1 and tx_2 and have timestamps 8 and 10, respectively. If tx_2 reads the value written by w_1 (from tx_1), then we must ensure that by the time tx_2's w_2 is replicated w_1 must have replayed on all replicas and will not be lost in f failures of $f+1$ replicas. Otherwise, since tx_2 and tx_1 may be handled by different cores, backups may potentially replay w_2 without replaying w_1. If the primary then fails, no backup (or the new primary) will have applied w_1 to the database. As a result, future transactions will only observe w_2, but not w_1, which violates serializability.

This shows that Phalanx must replicate w_1 and w_2 in their timestamp order to preserve the dependencies between them, e.g., w_2 depends on w_1.

Design insight. Phalanx selectively obeys the timestamp order. It replays and returns transactions in timestamp order *only when necessary*: when there exist dependencies among transactions. We say that transaction tx_2 depends on transaction tx_1 (denoted $tx_1 \Rightarrow tx_2$) if their requests access the same key(s), and at least one is an update. Since HCC (§4) guarantees that a dependent (tx_2) always has a larger timestamp than the transaction it depends on (tx_1), Phalanx can preserve dependencies by replaying them in their strictly increasing timestamp order. If a transaction does not depend on another, i.e., $tx_1 \not\Rightarrow tx_2$ and $tx_2 \not\Rightarrow tx_1$, Phalanx can replay them in any order, avoiding unnecessary stalls.

Frontline. Phalanx enables selective timestamp ordering by designing *Frontline*, a timestamp tracking technique. Each core on the primary keeps track of t_s, which is the timestamp of the most recent, safe transaction (log entry) in this core's sub-log. A transaction is safe if it has been appended to the logs of all backups. A core updates its t_s when it receives the *ack* of its propagated entries from the tail. The frontline t_f is the minimum t_s across all cores. t_f represents a threshold timestamp at which all transactions in the database with timestamps less than or equal to t_f are safe.

When a core of the primary begins propagating log entries, it piggybacks the current frontline t_f with them. Upon receiving both the entries and t_f (line 45), the backup core appends the new entries to its sub-log and continues propagating both the entries and t_f to the next backup (line 50). This backup then replays all entries whose timestamps are less than t_f, i.e., these entries (updates) are applied to the database (lines 51–54). Log replay is uninterruptible, that is, cores will not handle new entries until the current replay is complete. Replayed updates are then removed from the sub-log. The tail sends an *ack* to the primary when it has appended these new entries to its log (lines 47–48).

When the primary's core receives the *ack*, it updates its core-local safe time t_s and cross-core frontline t_f accordingly (lines 43–44). The primary periodically loops through the buffered response queue to release the responses of transactions whose timestamps are less than or equal to t_f (algorithm 5.3). These responses are released to their corresponding coordinators in their timestamp order.

When the transactions received by the primary happen to have monotonically increasing timestamps, t_s and t_f are advanced monotonically, similar to existing techniques. When the primary receives a new, out-of-order transaction whose timestamp is smaller than that of earlier transactions *and* the current t_f, the primary immediately lowers t_f and the corresponding core's t_s below the transaction's timestamp (lines 39–40). By immediately lowering t_f, Phalanx prevents the primary from prematurely releasing the responses of future transactions that may depend on this out-of-order transaction. That is, any future dependent transaction will only be released after the out-of-order transaction is finished. Algorithm 5.3 shows pseudocode for releasing buffered responses.

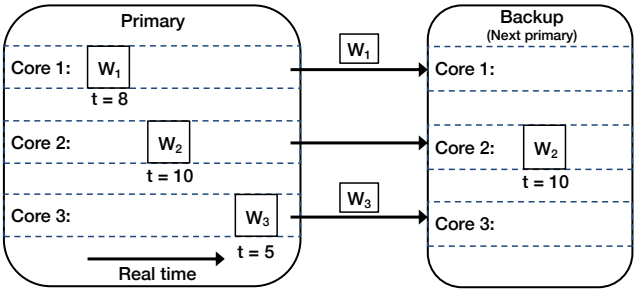


Figure 3: Writes operations w_1, w_2, w_3 from transactions tx_1, tx_2, and tx_3 (not shown), respectively, being replicated in each core's sub-log.

Buffering read responses. Although only updates are replicated to the backups, the primary must also buffer the responses of reads, including the reads in read-write transactions and read-only transactions. The primary handles these responses the same way that it handles update responses: it ensures that the updates observed by these reads have been stored on all replicas and cannot be lost in f failures.

Correctness. Phalanx guarantees that a transaction's response is released (i.e., this transaction is finished) only if its updates have been inserted in the logs of all backups and will eventually be replayed. For all transactions that have dependencies among them: HCC guarantees that they are timestamped in strictly increasing order, which reflects their dependencies. The frontline's forward movement returns them in this order. Otherwise, Phalanx does not block the replay of the current transaction tx_1 for the possible arrival of a later transaction tx_2 that has a smaller timestamp, thereby avoiding unnecessary stalls. This is safe because tx_1 must not depend on tx_2 as tx_1 was executed before tx_2, and tx_2 must not depend on tx_1 because tx_2's timestamp is smaller than tx_1's, thus it is safe to replay them in either order.

Failover. When the primary fails, the next live backup becomes the new primary. The new primary finishes replaying all sub-logs up to the frontline it knows, and then discards the remaining entries in its logs before servicing new requests. This is safe because the responses of these discarded transactions could not have been released to their coordinators. When the coordinators query the status of these discarded transactions, e.g., they have not received any responses for some time, the new primary replies with abort messages that make the coordinators abort these transactions on the servers.

6 Evaluation

We evaluate our system to answer the following questions:

- How well does TurboDB perform, compared to a representative distributed database, under skewed workloads?
- How well does TurboDB scale performance, specifically throughput, compared to the baseline?
- How well does TurboDB perform under different workloads with a variety of read-to-write ratios and levels of skew?

Implementation. We build TurboDB on CockroachDB [54] and Cicada [37], which are written in Go and C++, respectively. We change ∼4 K lines of Go in CockroachDB's codebase. We also employ Cicada's library to implement, replicate (i.e., implement Phalanx), and network (using gRPC [25]) a single-machine database in ∼10 K lines of C++. Of those, direct changes to Cicada's library are 5 lines long.

Baseline. We compare TurboDB with CockroachDB, which is a production distributed database that has been widely adopted by industry [34]. Its distributed concurrency control technique is a combination of timestamp-ordering and locking-based mechanisms, and it tolerates server failures with Raft. Our experiments have fault tolerance enabled for both TurboDB (Phalanx and Raft) and CockroachDB (Raft).

6.1 Experimental Setup

Workloads. We evaluate TurboDB under YCSB+T [17] and TPC-C [56]. YCSB+T contains one-shot key-value transactions, i.e., all requests are sent in one round in parallel as data locations are known a priori. Our experiments use the default parameters: 8 B key, 512 B value, 10 keys per transaction, and 95% reads. There are a total of 160 M keys. We vary the levels of skew by controlling the Zipfian constant (Zipf): uniform workloads have a Zipf of 0.01, medium-skewed workloads have a Zipf of 0.99 (∼8% of requests access the most popular key), and high-skewed workloads have a Zipf of 1.2 (∼25% of requests access the most popular key). We also include experiments that vary the read-to-write ratio.

TPC-C contains complex, multi-shot transactions, i.e., requests must be sent in multiple steps as data read in prior steps determines the read-/write-sets in later steps. TPC-C has five types of transactions: New Order, Payment, Delivery, Order Status, and Stock Level. We only implemented New Order which has the most complex transaction logic. We have 10 districts per warehouse and vary the level of skew by controlling the number of warehouses that are evenly distributed across machines. Our experiments show 64, 16, and 8 warehouses. The fewer warehouses, the more skewed the workload is.

Data placement. For the YCSB+T experiments, we promote hot data items to the turbo. We identify hot data items using a simple queries-per-second (QPS) count. It promotes as many data items to the turbo as can fit in its memory, taking care not to oversubscribe the turbo's CPU and memory capacity: 40 M keys of 160 M. We place the remaining, cool data items on CockroachDB, allowing its default data sharding schemes to balance the load. For the TPC-C experiments, we manually promote the two hottest tables—Warehouse and District—to the turbo. The remaining tables are partitioned by warehouse across the CockroachDB nodes.

Testbed. We run all experiments on CloudLab [22] in one data center. Each machine has 2.0 GHz CPUs with 8 physical (16 virtual) cores, 64 GB RAM, and a 10 Gbps network interface. For YCSB+T experiments, CockroachDB has 8 servers. Raft is run among these 8 servers instead of on a set of separate machines, as suggested by the CockroachDB technical team. TurboDB has 8 servers that handle the workloads. One of the 8 servers is the turbo, and the rest runs CockroachDB. TurboDB employs another 2 standalone machines as the backups, which do not directly handle the workload. Thus, TurboDB has a total of 10 server machines. The YCSB+T scalability experiments use up to 16 servers (18 servers for TurboDB). Similarly, the TPC-C experiments have 9 servers for CockroachDB and 11 servers for TurboDB.

An additional set of machines generate closed-loop client requests. Each experiment lasts 180 seconds. The first 120 warm up the system, e.g., the system shards the data. Performance metrics are collected during the remaining 60.

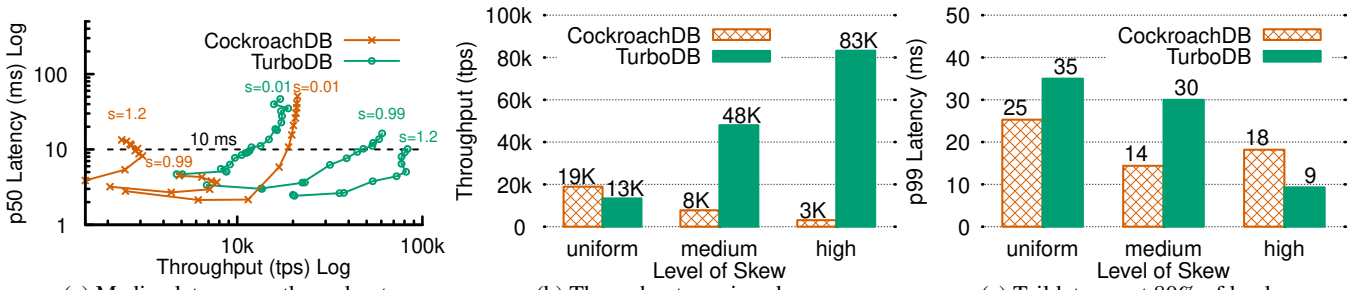
6.2 Result Overview

TurboDB outperforms CockroachDB by an order of magnitude higher throughput and 2× lower latency for YCSB+T, and by 1.6× higher throughput and lower latency for TPC-C, under skewed workloads. TurboDB scales out as well as CockroachDB under uniform workloads and shows much better scalability under medium and highly skewed workloads. TurboDB shows comparable performance to CockroachDB under uniform workloads and more significant performance improvements while skew increases. TurboDB consistently outperforms CockroachDB with different read-to-write ratios.

6.3 Latency & Throughput

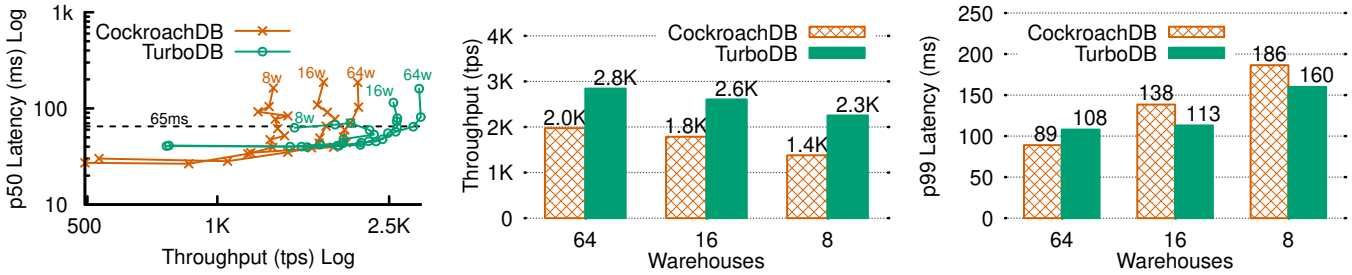
This section compares the performance of TurboDB and CockroachDB in terms of latency and throughput under YCSB+T and TPC-C workloads.

YCSB+T. Figure 4a plots the median latency vs. throughput graph for uniform, medium, and high skew as we increase load on the system. The dashed horizontal line shows a median latency of 10 ms, a reasonable operating point. TurboDB consistently outperforms CockroachDB under medium and high skew, due to its turbo being able to efficiently execute contended transactions. For instance, TurboDB has more than 4× lower abort rate than CockroadhDB with a Zipf of 1.2. Although TurboDB is not designed for uniform workloads,



(a) Median latency vs. throughput (b) Throughput varying skew (c) Tail latency at 80% of load

Figure 4: Latency (medium and tail) and throughput of TurboDB and CockroachDB under YCSB+T workloads with varying levels of skew (s indicates the Zipfian constant.



(a) Median latency vs. throughput (b) Throughput varying skew (c) Tail Latency at 80% of load

Figure 5: Latency (medium and tail) and throughput of TurboDB and CockroachDB under TPC-C New Order workloads with varying levels of skew, controlled by the number of warehouses (w).

its performance is comparable to (slightly worse than) CockroachDB for these workloads, due to the design choices TurboDB makes specifically for skewed workloads, e.g., finale commit and chain-shaped replication may increase latency.

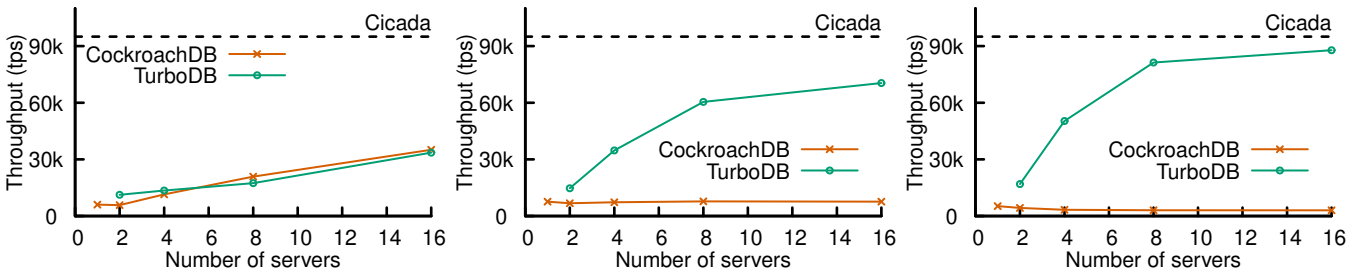
Figure 4b shows the throughput while varying the level of skew at the operating point, i.e., a median latency of 10 ms. TurboDB exhibits more significant throughput advantages when skew increases, e.g., up to 17× improvements. This is because a higher level of skew causes more transactions to abort on CockroachDB, while TurboDB can reduce aborts with its local concurrency control and execute these transactions more quickly with its one-stop execution. Moreover, TurboDB's throughput increases when skew increases because more requests can benefit from the turbo, while the performance of traditional databases often keeps getting worse when the workload becomes more skewed.

Figure 4c shows the tail (p99) latency when both systems operate at 80% of their maximum load. TurboDB exhibits slightly higher latency on uniform and medium skewed workloads. This is because HCC's serial finale commit and Phalanx's chain propagation increase latency, and this affects the tail of the distribution more significantly. However, this latency impact is offset by the latency improvements under high skew workloads where TurboDB has 2× lower tail latency.

TPC-C New Order. Figure 5 shows the performance under TPC-C New Order workloads, which exhibits a similar takeaway of performance improvements enabled by TurboDB while the improvements are not as significant as that for YCSB+T. This is because TPC-C transactions are multi-shot and have much more complex logic than YCSB+T, and because it is non-trivial to partition TPC-C and find the right popular data items to store on the turbo. Our experiments make the turbo store popular District tables, and we expect even greater performance improvements with careful partitioning. That said, TurboDB achieves consistently better performance under medium and high skew, as shown in Figure 5a. TurboDB achieves consistently higher throughput (up to 1.65× higher) with low (64 warehouses), medium (16 warehouses), and high skew (8 warehouses), as shown in Figure 5b. TurboDB has lower tail latency for medium and high skew (up to 1.5× lower), and slightly higher tail latency when skew is low. The low skew setting (64 warehouses) still exhibits some contention and is far from being uniform, and TurboDB's performance improvements may diminish when the number of warehouses is sufficiently large.

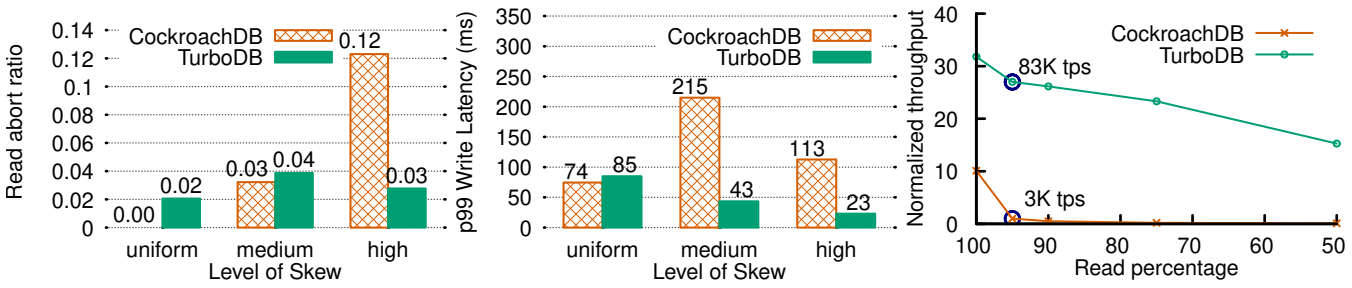
6.4 Scalability

Figure 6 shows peak throughput of TurboDB and CockroachDB under YCSB+T when we increase the number of machines (and the amount of data stored) in the system. Figure 6a shows that TurboDB can scale as linearly as CockroachDB under uniform workloads for which TurboDB is not designed specifically. This shows that the overhead of TurboDB's design under uniform workloads does not much



(a) Uniform ($s = 0.01$) (b) Medium skew ($s = 0.99$) (c) High skew ($s = 1.2$)

Figure 6: Scalability under YCSB+T read-heavy (95% reads) workloads while increasing the number of servers and the amount of data. The dashed line denotes the peak throughput of networked, replicated Cicada (the turbo).



(a) YCSB+T Abort ratio (b) p99 latency of write-only txns (c) Varying Read Fraction (s=1.2)

Figure 7: Additional experiments that report abort ratios, write tail latency, and throughput with different read fractions.

affect the system's overall performance, because the workload is "easy," i.e., uniform, and additional overhead is affordable.

TurboDB scales throughput significantly better than CockroachDB under medium and highly skewed workloads, as shown in Figure 6b and Figure 6c. CockroachDB does not scale under skewed workloads because it is bottlenecked on processing contended transactions that access a small set of popular keys. Adding more machines does not address the bottleneck. In contrast, TurboDB scales throughput linearly until the turbo reaches its capacity, i.e., when networked, replicated Cicada is running at maximum throughput.

TurboDB's throughput approaches that of networked, replicated Cicada, denoted by the dotted horizontal line, with a sufficient number of servers. TurboDB can sustain its throughput and scales no worse than traditional distributed databases after Cicada becomes the bottleneck. Moreover, TurboDB scales up faster at higher skew because more transactions can leverage the performance multipliers of the turbo.

6.5 Additional Experiments

We show more experiments with YCSB+T workloads.

Abort ratio. An important source of improvements enabled by the turbo is the reduction of aborts. (Another source is the fast execution of contended transactions, enabled by one-stop execution.) Figure 7a plots the abort ratio which is the number of the transactions that were ever aborted to the number of committed transactions, under uniform, medium, and high skew. While TurboDB has abort ratios similar to CockroachDB under uniform and medium skewed workloads, TurboDB has a 4× lower abort ratio under high skew, which is enabled by the performance multipliers of the turbo.

Write latency. To fully understand TurboDB's performance improvements, we examine the latency of write-only transactions from Figure 4b's workload. We focus on write requests because they lead to conflicts and because they reflect the costs of replication as only writes are replicated through Phalanx. Figure 7b shows the tail (p99) latency of write-only transactions at different levels of skew. TurboDB has significantly lower write latency at medium and high skew because the turbo can execute these writes with fewer aborts and because one-stop execution avoids cross-phase locking, while CockroachDB requires distributed locks (i.e., write intents) that significantly prolongs the execution time.

Varying read fractions. Figure 7c shows the throughput of both systems when we vary the read-to-write ratio of the highly skewed YCSB+T workloads (Zipf of 1.2). We normalize throughput against the maximum throughput under the default setting of 95% reads. Both TurboDB and CockroachDB have lower throughput with more writes, because write operations are more costly compared to reads and because writes increase the likelihood of conflicts, and TurboDB has consistently higher throughput than CockroachDB.

7 Related Work

TurboDB builds on earlier work on single-machine databases, distributed databases, partitioning techniques, and replication techniques. Section 5 discussed replication techniques. We now review each of the other categories in turn.

Single-machine databases. There exists a large body of research on designing and building fast single-machine databases [23, 24, 29, 31, 33, 37, 47, 58]. Because their data resides on one machine, these databases capitalize on readily available global views of the system to employ sophisticated optimizations (see discussion in §2). In contrast, in distributed databases, global views are non-trivial to construct.

TurboDB's design makes it possible for these optimizations to be used in distributed systems. What's more, TurboDB improves its performance while retaining the distributed system's ability to scale system capacity to support large-scale applications—ones that would have been too large to fit in a single-machine database.

We highlight the constraint that TurboDB requires the turbo (single-machine database) to employ a timestamp-based concurrency control. That said, many existing databases would make good candidates [29, 33, 37, 58].

Distributed databases and systems. There also exists a large body of work on distributed databases and distributed systems [12, 49, 52, 54, 55, 65]. Some achieve good performance by constraining operations of a transaction to access the same logical partition [11, 13]. These systems rely on careful data partitioning, which may be challenging for today's complex applications to achieve. In contrast, TurboDB requires no partitioning constraints.

Some other systems trade off strong consistency for better performance [11, 38, 39, 42, 50]. Unfortunately, weakly consistent transactions complicate application development, yet, are still subject to performance degradation under skewed workloads. In contrast, TurboDB provides strong consistency (process-ordered serializability).

Some other systems support restricted transaction models, e.g., read-only and/or write-only transactions [21, 38, 39]. Unfortunately, this complicates application development. In contrast, TurboDB supports general transactions.

Sequencers and RDMA. Some systems leverage a centralized component, e.g., a sequencer or a shared log, to serialize all transactions [5, 6, 30, 35, 36, 55, 60]. These techniques' insights bears similarity to TurboDB's, i.e., leveraging a powerful centralized entity to tackle the most challenging problems in the system's design. However, unlike TurboDB, they require that all transactions pass through the sequencer. Instead, TurboDB only forwards a fraction of (popular) keys through the turbo, enabling scalability with less internal complexity and fewer resources than recent scalable sequencer-based systems [18, 27].

Some systems leverage specialized hardware and network abstractions [20], e.g., RDMA and DPDK [19]. These networking optimizations are orthogonal to TurboDB's performance improvements, as TurboDB can also adopt and benefit from them.

Partitioning techniques. One line of work handles contention by partitioning the keyspace by workload access patterns, i.e., keys that are likely accessed together by transactions are co-located on the same machine [15, 45, 48, 53, 54, 57, 61, 63]. These works aim to reduce the number of nodes that each transaction must contact—ideally, only one—such that they are as "non-distributed" as possible. For instance, Chiller [64] and Quro [61] co-locate keys that are both popular and often accessed together on the same machine.

While these partitioning techniques benefit workloads where transactions minimally access keys on different machines, their benefits diminish when no such obvious groups of keys exist. In contrast, TurboDB's performance benefits are agnostic to whether the keyspace is partitionable. Even when transactions access both the turbo and the servers, TurboDB's HCC and Phalanx ensure such transactions benefit from the turbo's performance multipliers.

8 Conclusion

Distributed databases are challenged by skewed workloads, which are common in real-world applications. These workloads cause high contention, which are exacerbated by network latencies. TurboDB presents a novel hybrid architecture that integrates a single-machine database in a distributed database to "turbocharge" its overall performance under skewed workloads. TurboDB leverages the single-machine database's performance multipliers to efficiently execute contended transactions. It introduces new designs, HCC and Phalanx, that tackle the challenges of concurrency control and replication under its hybrid architecture. Consequently, TurboDB achieves up to an order of magnitude better performance than a representative distributed database.

Acknowledgments. We thank our shepherd, Dan Ports, and our anonymous reviewers for their invaluable feedback. We also thank Christopher Hodsdon, Yue Tan, Samuel Ginzburg, Mike Wong, and Gongqi Huang for their helpful suggestions. This material is based upon work supported by the National Science Foundation under Grant No. CNS 2241719, CNS 1824130, CNS 2327609, and CNS 2321724. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Availability. Code and experimental scripts are available at https://github.com/princeton-sns/TurboDB.

References

[1] Marcos K Aguilera, Tudor David, Rachid Guerraoui, and Junxiong Wang. Locking timestamps versus locking objects. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 2018.

[2] Phillipe Ajoux, Nathan Bronson, Sanjeev Kumar, Wyatt Lloyd, and Kaushik Veeraraghavan. Challenges to adopting stronger consistency at scale. In *Workshop on Hot Topics in Operating Systems (HotOS)*, 2015.

[3] Timothy G Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. Linkbench: A database benchmark based on the facebook social graph. In *ACM International Conference on Management of Data (SIGMOD)*, pages 1185–1196, 2013.

[4] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS and PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, pages 53–64, 2012.

[5] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobbler, Michael Wei, and John D Davis. CORFU: A shared log design for flash clusters. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 1–14, 2012.

[6] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D Davis, Sriram Rao, Tao Zou, and Aviad Zuck. Tango: Distributed data structures over a shared log. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 325–340, 2013.

[7] Philip A Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR)*, 13(2):185–221, 1981.

[8] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and zipf-like distributions: Evidence and implications. In *IEEE International Conference on Computer Communications (INFOCOMM)*, volume 1, pages 126–134. IEEE, 1999.

[9] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. TAO: Facebook's distributed data store for the social graph. In *USENIX Annual Technical Conference (ATC)*, pages 49–60, 2013.

[10] Audrey Cheng, Xiao Shi, Aaron Kabcenell, Shilpa Lawande, Hamza Qadeer, Jason Chan, Harrison Tin, Ryan Zhao, Peter Bailis, Mahesh Balakrishnan, Nathan Bronson, Natacha Crooks, and Ion Stoica. TAOBench: An end-to-end benchmark for social network workloads. *Proceedings of the VLDB Endowment (PVLDB)*, 15(9):1965–1977, 2022.

[11] Brian F Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment (PVLDB)*, 1(2):1277–1288, 2008.

[12] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ 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. Spanner: Google's globally distributed database. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 261–264, 2012.

[13] CosmosDB. CosmosDB. https://azure.microsoft.com/en-us/services/cosmos-db/, 2022.

[14] James Cowling and Barbara Liskov. Granola: Low-overhead distributed transaction coordination. In *USENIX Annual Technical Conference (ATC)*, Jun 2012.

[15] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. Schism: A workload-driven approach to database replication and partitioning. *Proceedings of the VLDB Endowment (PVLDB)*, 3(1–2):48–57, Sep 2010.

[16] Khuzaima Daudjee and Kenneth Salem. Lazy database replication with ordering guarantees. In *IEEE International Conference on Data Engineering (ICDE)*, 2004.

[17] Akon Dey, Alan Fekete, Raghunath Nambiar, and Uwe Röhm. YCSB+T: Benchmarking web-scale transactional databases. In *IEEE International Conference on Data Engineering (ICDE) Workshops*, pages 223–230, 2014.

[18] Cong Ding, David Chu, Evan Zhao, Xiang Li, Lorenzo Alvisi, and Robbert Van Renesse. Scalog: Seamless reconfiguration and total order in a scalable shared log. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 325–338, 2020.

[19] DPDK. DPDK. http://dpdk.org/, 2020.

[20] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. FaRM: Fast remote memory. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2014.

[21] Jiaqing Du, Sameh Elnikety, Amitabha Roy, and Willy Zwaenepoel. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *ACM Symposium on Cloud Computing (SoCC)*, 2013.

[22] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *USENIX Annual Technical Conference (ATC)*, pages 1–14, 2019.

[23] Jose M Faleiro and Daniel J Abadi. Rethinking serializable multiversion concurrency control. *Proceedings of the VLDB Endowment (PVLDB)*, 8(11):944–955, 2015.

[24] Jose M. Faleiro, Daniel J. Abadi, and Joseph M. Hellerstein. High performance transactions via early write visibility. *Proceedings of the VLDB Endowment (PVLDB)*, 10(5), 2017.

[25] gRPC. gRPC, 2020. https://github.com/grpc.

[26] Zhenyu Guo, Chuntao Hong, Mao Yang, Dong Zhou, Lidong Zhou, and Li Zhuang. Rex: Replication at the speed of multi-core. In *European Conference on Computer Systems (EuroSys)*, pages 1–14, 2014.

[27] Christopher Hodsdon, Theano Stavrinos, Ethan Katz-Bassett, and Wyatt Lloyd. Mason: Scalable, contiguous sequencing for building consistent services. *Journal of Systems Research (JSys)*, 3(1), 2023.

[28] Qi Huang, Ken Birman, Robbert van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C. Li. An analysis of facebook photo caching. In *ACM Symposium on Operating Systems Principles (SOSP)*, November 2013.

[29] Yihe Huang, William Qian, Eddie Kohler, Barbara Liskov, and Liuba Shrira. Opportunities for optimism in contended main-memory multicore transactions. *Proceedings of the VLDB Endowment (PVLDB)*, 13(5):629–642, Jan 2020.

[30] Anuj Kalia, Michael Kaminsky, and David G Andersen. Design guidelines for high performance RDMA systems. In *USENIX Annual Technical Conference (ATC)*, pages 437–450, 2016.

[31] 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. H-store: A high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment (PVLDB)*, 1(2):1496–1499, Aug 2008.

[32] Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. All about eve: Execute-verify replication for multi-core servers. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 237–250, 2012.

[33] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. Ermia: Fast memory-optimized database system for heterogeneous workloads. In *ACM International Conference on Management of Data (SIGMOD)*, pages 1675–1687, 2016.

[34] Cockroach Labs. CockroachDB, 2020. https://www.cockroachlabs.com/.

[35] Jialin Li, Ellis Michael, and Dan RK Ports. Eris: Coordination-free consistent transactions using in-network concurrency control. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 104–120, 2017.

[36] Jialin Li, Ellis Michael, Naveen Kr Sharma, Adriana Szekeres, and Dan RK Ports. Just say NO to paxos overhead: Replacing consensus with network ordering. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 467–483, 2016.

[37] Hyeontaek Lim, Michael Kaminsky, and David G Andersen. Cicada: Dependably fast multi-core in-memory transactions. In *ACM International Conference on Management of Data (SIGMOD)*, pages 21–35, 2017.

[38] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 401–416, 2011.

[39] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Stronger semantics for low-latency geo-replicated storage. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 313–328, 2013.

[40] Haonan Lu, Christopher Hodsdon, Khiem Ngo, Shuai Mu, and Wyatt Lloyd. The SNOW theorem and latency-optimal read-only transactions. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 135–150, 2016.

[41] Haonan Lu, Shuai Mu, Siddhartha Sen, and Wyatt Lloyd. NCC: Natural concurrency control for strictly serializable datastores by avoiding the timestamp-inversion pitfall. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 305–323, 2023.

[42] MySQL. MySQL 5.6 Reference Manual. https://dev.mysql.com/doc/refman/5.6/en/, 2020.

[43] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference (ATC)*, pages 305–319, 2014.

[44] Christos H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, October 1979.

[45] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *ACM International Conference on Management of Data (SIGMOD)*, pages 61–72, 2012.

[46] Dai Qin, Angela Demke Brown, and Ashvin Goel. Scalable replay-based replication for fast databases. *Proceedings of the VLDB Endowment (PVLDB)*, 10(13):2025–2036, 2017.

[47] Dai Qin, Angela Demke Brown, and Ashvin Goel. Caracal: Contention management with deterministic concurrency control. In *ACM Symposium on Operating Systems Principles (SOSP)*, page 180–194, 2021.

[48] Abdul Quamar, K Ashwin Kumar, and Amol Deshpande. Sword: scalable workload-aware data placement for transactional workloads. In *International Conference on Extending Database Technology (EDBT)*, pages 430–441, 2013.

[49] Kun Ren, Dennis Li, and Daniel J Abadi. SLOG: serializable, low-latency, geo-replicated transactions. *Proceedings of the VLDB Endowment (PVLDB)*, 12(11):1747–1761, 2019.

[50] riak. riak, 2022. https://riak.com/index.html.

[51] Weihai Shen, Ansh Khanna, Sebastian Angel, Siddhartha Sen, and Shuai Mu. Rolis: A software approach to efficiently replicating multi-core transactions. In *European Conference on Computer Systems (EuroSys)*, pages 69–84, 2022.

[52] Swaminathan Sivasubramanian. Amazon DynamoDB: A seamlessly scalable non-relational database service. In *ACM International Conference on Management of Data (SIGMOD)*, pages 729–730, 2012.

[53] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proceedings of the VLDB Endowment (PVLDB)*, 8(3):245–256, 2014.

[54] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruner, Justin Jaffray, Lucy Zhang, and Peter Mattis. CockroachDB: The resilient geo-distributed sql database. In *ACM International Conference on Management of Data (SIGMOD)*, pages 1493–1509, 2020.

[55] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *ACM International Conference on Management of Data (SIGMOD)*, pages 1–12, 2012.

[56] TPC. TPC-C: An on-line transaction processing benchmark. http://www.tpc.org/tpcc/, 2020.

[57] Khai Q Tran, Jeffrey F Naughton, Bruhathi Sundarmurthy, and Dimitris Tsirogiannis. JECB: A join-extension, code-based approach to OLTP data partitioning. In *ACM International Conference on Management of Data (SIGMOD)*, pages 39–50, 2014.

[58] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 18–32, 2013.

[59] Robbert Van Renesse and Fred B Schneider. Chain replication for supporting high throughput and availability. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 91–104, 2004.

[60] Michael Wei, Amy Tai, Christopher J Rossbach, Ittai Abraham, Maithem Munshed, Medhavi Dhawan, Jim Stabile, Udi Wieder, Scott Fritchie, Steven Swanson, et al. vCorfu: A cloud-scale object store on a shared log. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 35–49, 2017.

[61] Cong Yan and Alvin Cheung. Leveraging lock contention to improve OLTP application performance. *Proceedings of the VLDB Endowment (PVLDB)*, 9(5):444–455, 2016.

[62] Juncheng Yang, Yao Yue, and KV Rashmi. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *USENIX Symposium on Operating Systems*

Design and Implementation (OSDI), pages 191–208, 2020.

[63] Erfan Zamanian, Carsten Binnig, and Abdallah Salama. Locality-aware partitioning in parallel database systems. In *ACM International Conference on Management of Data (SIGMOD)*, pages 17–30, 2015.

[64] Erfan Zamanian, Julian Shun, Carsten Binnig, and Tim Kraska. Chiller: Contention-centric transaction execution and data partitioning for modern networks. In *ACM International Conference on Management of Data (SIGMOD)*, pages 511–526, 2020.

[65] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. Building consistent transactions with inconsistent replication. *ACM Transactions on Computer Systems (TOCS)*, 35(4):12:1–12:37, 2018.

[66] ZooKeeper. Apache ZooKeeper. https://zookeeper.apache.org/, 2022.