Taking Omid to the Clouds: Fast, Scalable Transactions for Real-Time Cloud Analytics

Ohad Shacham
Yahoo Research
ohads@oath.com

Yonatan Gottesman
Yahoo Research
yonatang@oath.com

Aran Bergman
Technion
aranb@campus.technion.ac.il

Edward Bortnikov
Yahoo Research
ebortnik@oath.com

Eshcar Hillel
Yahoo Research
eshcar@oath.com

Idit Keidar
Technion and Yahoo Research
idish@ee.technion.ac.il

ABSTRACT

We describe how we evolve Omid, a transaction processing system for Apache HBase, to power Apache Phoenix, a cloud-grade real-time SQL analytics engine.

Omid was originally designed for data processing pipelines at Yahoo, which are, by and large, throughput-oriented monolithic NoSQL applications. Providing a platform to support converged real-time transaction processing and analytics applications – dubbed *translytics* – introduces new functional and performance requirements. For example, SQL support is key for developer productivity, multi-tenancy is essential for cloud deployment, and latency is cardinal for just-in-time data ingestion and analytics insights.

We discuss our efforts to adapt Omid to these new domains, as part of the process of integrating it into Phoenix as the transaction processing backend. A central piece of our work is latency reduction in Omid's protocol, which also improves scalability. Under light load, the new protocol's latency is 4x to 5x smaller than the legacy Omid's, whereas under increased loads it is an order of magnitude faster. We further describe a *fast path* protocol for single-key transactions, which enables processing them almost as fast as native HBase operations.

PVLDB Reference Format:
Shacham, Gottesman, Bergman, Bortnikov, Hillel, and Keidar. Taking Omid to the Clouds:
Fast, Scalable Transactions for Real-Time Cloud Analytics. *PVLDB*, 11(12): xxxx-yyyy, 2018.
DOI: https://doi.org/10.14778/3229863.3229868

1. INTRODUCTION

In recent years, transaction processing [35] technologies have paved their way into multi-petabyte big data platforms [39, 26, 40]. Modern industrial *transaction processing systems (TPSs)* [39, 40, 14, 8] complement existing underlying NoSQL key-value storage with *atomicity*, *consistency*, *isolation* and *durability* (ACID) semantics that enable programmers to perform complex data manipulation without over-complicating their applications. Google's Percolator [39] pioneered a transaction API atop the Bigtable storage. Apache Incubator projects Tephra [14] and Omid [40] followed suit with Apache HBase [2].

Such technologies must evolve in light of the recent shift towards massive deployment of big data services in public clouds; for example, the AWS cloud can run HBase as part of Elastic MapReduce [1], and SQL engines like Apache Phoenix increasingly target public cloud deployment. Yet adapting TPS technology for cloud use is not without challenges, as we now highlight.

Proceedings of the VLDB Endowment, Vol. 11, No. 12

DOI: https://doi.org/10.14778/3229863.3229868

1.1 Challenges

Diverse functionality. Large-scale transaction support was initially motivated by specific use cases like content indexing for web search [39, 40] but rapidly evolved into a wealth of OLTP and analytics applications (e.g., [41]). Today, users expect to manage diverse workloads within a single data platform, to avoid lengthy and error-prone extract-transform-load processes. A Forrester report [15] coins the notion of *translytics* as "a unified and integrated data platform that supports multi-workloads such as transactional, operational, and analytical simultaneously in real-time, ... and ensures full transactional integrity and data consistency".

As use cases become more complex, application developers tend to prefer high-level SQL abstractions to crude NoSQL data access methods. Indeed, scalable data management platforms (e.g., Google Spanner [26], Apache Phoenix [6], and CockroachDB [8]) now provide full-fledged SQL interfaces to support complex query semantics in conjunction with strong data guarantees. SQL APIs raise new requirements for transaction management, e.g., in the context of maintaining secondary indexes for accelerated analytics.

Scale. Public clouds are built to serve a multitude of applications with elastic resource demands, and their efficiency is achieved through scale. Thus, cloud-first data platforms are designed to scale well beyond the limits of a single application. For example, Phoenix is designed to scale to 10K query processing nodes in one instance, and is expected to process hundreds of thousands or even millions of transactions per second (*tps*).

Latency. Similarly to many technologies, the adoption of transactions took a "functionality-first" trajectory. For

example, the developers of Spanner [26] wrote: "We believe it is better to have application programmers deal with performance problems due to overuse of transactions as bottlenecks arise, rather than always coding around the lack of transactions". Yet the expectation for low latency is rapidly picking up. Whereas early applications of big data transaction processing systems were mostly throughput-sensitive [39, 40], with the thrust into new interactive domains like social networks [7], messaging [22] and algorithmic trading [11] latency becomes essential. SLAs for interactive user experience mandate that simple updates and point queries complete within single-digit milliseconds. The early experience with Phoenix shows that programmers often refrain from using transaction semantics altogether for fear of high latency, and instead adopt solutions that may compromise consistency, e.g., causing an index to inconsistently represent the data.

Multi-tenancy. Data privacy is key in systems that host data owned by multiple applications. Maintaining access rights is therefore an important design consideration for TPSs, e.g., in maintaining shared persistent state.

1.2 Evolving Omid for public clouds

The motivation of this work is to improve scalability, latency and functionality of Omid [5] to the scale needed in large, multi-tenant clouds. The paper describes recent functionality extensions and performance improvements we made in Omid to power Phoenix [6], a Hadoop SQL-compliant data analytics platform designed for cloud deployment.

In contrast to earlier SQL query engines for Hadoop (e.g., Hive[3] and Impala [4]), which focused on large-scale processing of immutable data, Phoenix targets converged data ingestion and analytics [16], which necessitates transaction semantics. Phoenix applications are both latency and throughput sensitive. Phoenix uses HBase as its key-value storage layer, and Omid as a transaction processing layer atop HBase[1].

Omid is an open source transaction manager for HBase. It provides a (somewhat relaxed) variant of *snapshot isolation (SI)* [21], similarly to other modern transaction managers [39, 26, 14, 8]. Its implementation is lock-free, which makes it scale better than traditional lock-based serializability protocols. Omid is database-neutral, i.e., it can use any data store with a traditional NoSQL API (see Section 2). In particular, it does not require any changes to HBase code.

This paper makes the following contributions:

1. *Protocol re-design for low-latency* (Section 3). The new protocol, *Omid Low Latency (Omid LL)*, dissipates Omid's major architectural bottleneck. It reduces the latency of short transactions by 5x under light load, and by 10x–100x under heavy load. It also scales the overall system throughput to 550K tps while remaining within real-time latency SLAs. In contrast to previously published protocols (e.g., [39]), our solution is amenable to multi-tenancy. The development of this new feature is reported in [10].
2. *Novel fast-path (FP) algorithm* (Section 4). Our novel Omid FP protocol maximizes the performance of single-key transactions that arise in many production use cases. It defines a dedicated API for single-key read, write, and read-modify-write transactions, which execute at the latency of native HBase operations regardless of system load. They run twice as fast as Omid LL transactions, and complete within 2–4 ms on mid-range hardware. This comes at the cost of a minor (15–25%) negative impact on long (latency-insensitive) transactions. Note that Omid LL and Omid FP are complementary latency-oriented improvements.
3. *Evaluation and comparison with alternative designs* (Section 5). We extensively evaluate the new algorithms. We further compare Omid's centralized architecture to a decentralized *two-phase commit (2PC)*-based approach, as adopted in Percolator and CockroachDB.
4. *SQL compliance* (Section 6). We add support for creating secondary indexes on-demand, without impeding concurrent database operations or sacrificing consistency. We further extend the traditional SI model, which provides single-read-point-single-write-point semantics, with multiple read and write points. This functionality is used to avoid recursive read-your-own-writes scenarios in complex queries. The development of these new features in Omid and Phoenix is reported in [9] and [12], respectively.

Finally, Section 7 reviews related work, and Section 8 concludes.

2. BACKGROUND

A transaction processing system runs atop an underlying key-value store and allows users to bundle multiple data operations into a single atomic transaction. Section 2.1 describes the NoSQL data model and the data access API of the underlying key-value store, and Section 2.2 defines transaction semantics provided by the TPS. Section 2.3 provides background on the modus operandi of existing TPSs that support SI, including Omid. Finally, Section 2.4 overviews HBase and Phoenix.

2.1 Data store API

The data store holds *objects* (often referred to as *rows* or *records*) identified by unique *keys*. Each row can consist of multiple *fields*, representing different *columns*. We consider multi-versioned objects, where object values are associated with *version numbers*, and multiple versions associated with the same key may co-exist. We further assume that a write operation can specify the version number it writes to. The data store provides the following API:

get(key, version) – returns the requested version of key. The API further allows traversing (reading) earlier versions of the same key.

scan (fromKey, toKey, version) – a range query (extension of get). Returns an iterator that supports retrieval of multiple records using getNext() calls.

put(key, version, fields, values) – creates or updates an object, setting the specified fields to the specified values. If the version already exists, its value is updated; otherwise, a new version is added.

remove(key, version) – removes an object with the given key and version.

check&mutate(key, field, old, new) – checks the record associated with key. If field holds old, replaces it with

[1]Tephra is supported as alternative transaction manager, though its scalability and reliability are inferior to Omid's.

new; either way returns field's previous value. If old is nil, creates field and initializes it with new only if key is missing in data store.

Most NoSQL data store implementations guarantee atomic execution of each of the above API's, except traversals of iterators returned by scan.

2.2 Transaction semantics

TPSs provide *begin* and *commit* APIs for delineating transactions: a *transaction* is a sequence of *read* and *write* operations on different objects that occur between begin and commit. For simplicity, we only describe single-key operations here; multi-key range queries (scans) are treated as sequences of reads. Thus, transactional reads and writes are implemented using the datastore's get and put operations. Two transactions are said to be *concurrent* if their executions overlap, i.e., one of them begins between the begin time and commit time of the other; otherwise, we say that they are *non-overlapping*.

A TPS ensures the ACID properties for transactions: *atomicity* (all-or-nothing), *consistency* (preserving each object's semantics), *isolation* (in that concurrent transactions do not see each other's partial updates), and *durability* (whereby updates survive crashes).

Different isolation levels can be considered for the third property. We consider a variant of *snapshot isolation (SI)* [21] that, similarly to *generalized snapshot isolation* [30], relaxes the real-time order requirement. Nevertheless, our implementation only relaxes the ordering of fast path transactions (described in Section 4) relative to regular ones (that do not use the fast path); regular transactions continue to satisfy SI amongst themselves. Moreover, Omid LL, without the fast path, satisfies SI as Omid does.

Our relaxed correctness condition satisfies the key "snapshot" property of SI, which ensures that a transaction reading from the database does not see a mix old and new values. For example, if a transaction updates the values of two stocks, then no other transaction may observe the old value of one of these stocks and the new value of the other. However, it relaxes the real-time order guarantee of SI by allowing (fast-path) transactions to take effect 'in the past'. Specifically, the system enforces a total order $\mathcal{T}$ on all committed transactions, so that (i) non-overlapping transactions that update the same key occur in $\mathcal{T}$ in order of their commit times; (ii) each transaction's read operations see a consistent snapshot of the database reflecting a prefix of $\mathcal{T}$; and (iii) a transaction commits only if none of the items it updates is modified by a transaction ordered in $\mathcal{T}$ after its snapshot time and before its commit time.

Note that as with SI, two concurrent transactions conflict only if they both *update* the same item. In contrast, under serializability, a transaction that updates an item also conflicts with transactions that *read* that item. Snapshot isolation is thus amenable to implementations (using multi-versioning) that allow more concurrency than serializable ones, and hence scale better. It is provided by popular database technologies such as Oracle, PostgreSQL, and SQL Server, and TPSs such as Percolator, Omid, and CockroachDB.

Following a commit call, the transaction may successfully *commit*, whereby all of its operations take effect, or *abort*, in which case none of its changes take effect.

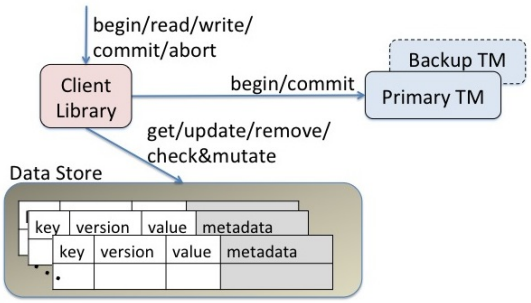


Figure 1: Transaction processing architecture: A client library exposes an API for executing transactions of data store operations. A centralized Transaction Manager (TM) handles transaction begin and commit requests, while data is written directly to the underlying data store. The TM has a backup for high availability.

Algorithm 1 TPS operation schema.

```
1:  procedure BEGIN
2:      obtain read timestamp ts_r
3:  procedure WRITE(ts_r, key, fields, values)
4:                                          ▷ transactional write
5:      optionally check for conflicts and abort if found
6:      indicate write intent for key with values and ts_r
7:      add key to local write-set
8:  procedure READ(ts_r, key)               ▷ transactional read
9:      if key has write intent then
10:         resolve, possibly abort writing transaction
11:     return highest version ≤ ts_r of key
12: procedure COMMIT(ts_r, write-set)
                                  ▷ check for write-write conflicts
13:     obtain commit timestamp ts_c
14:     if validate(write-set, ts_r) then
15:         write commit version ts_c to persistent commit entry
16:     else
17:         abort
18:     post-commit: update meta-data
```

2.3 TPS operation schema

Figure 1 depicts, at a high level, the primary components of the TPS architecture, their APIs, and interaction with the data store.

In many TPSs, transaction processing follows the following general schema, outlined in Algorithm 1, while systems vary in their implementations of each of the steps.

Most of the systems employ a centralized *transaction manager (TM)* service [39, 34, 40, 14], sometimes called timestamp oracle, for timestamp allocation and other functionalities. Because a centralized service can become a single point of failure, the TM is sometimes implemented as a primary-backup server pair to ensure its continued availability following failures.

Begin. When a transaction begins, it obtains a read timestamp (version) ts_r for reading its consistent snapshot. In most cases, this is done using the centralized TM [39, 34, 40, 14].

Transactional writes. During a transaction, a write operation indicates its *intent* to write to a single object a certain new value with a certain version number. In Omid, the version is the transaction's ts_r, which exceeds all versions written by transactions that committed before the current

transaction began. Note that the version order among concurrent transactions that attempt to update the same key is immaterial, since all but one of these transactions are doomed to abort.

It is possible to buffer write intents locally (at the client) in the course of the transaction, and add the write intents to the data store at commit time [39].

In some solutions writes check for conflicts before declaring their intents [8], whereas in others, all conflict detection is deferred to commit time [39, 34, 40, 14].

Transactional reads. The reads of a given transaction obtain a consistent snapshot of the data store at logical time (i.e., version) ts_r. Each read operation retrieves the value of a single object associated with the highest timestamp that is smaller or equal to the transaction's ts_r.

On encountering a write intent, read cannot proceed without determining whether the tentative write should be included in its snapshot, for which it must know the writing transaction's commit status. To this end, TPSs keep per-transaction *commit entries*, which are the source of truth regarding the transaction status (pending, committed, or aborted). This entry is updated in line 15 of Algorithm 1 as we explain below, and is checked in order to resolve write intents in line 10. In some cases [39, 8], when the status of the writing transaction is undetermined, the read forcefully aborts it by updating the commit entry accordingly, as explained below.

Commit. Commit occurs in four steps:

1. Obtain a commit timestamp, ts_c. In most cases, e.g., [39, 34, 40, 14], this is the value of some global clock maintained by a centralized entity.
2. *Validate* that the transaction does not conflict with any concurrent transaction that has committed since it had begun. For SI, we need to check for write-write conflicts only. If write intent indications are buffered, they are added at this point [39]. Validation can be centralized [34, 40, 14] or distributed [39, 8].
3. *Commit* or abort in one irrevocable atomic step by persistently writing to the *commit entry*, which can reside in a global table [40, 8] or alongside the first key written by the transaction [39].
4. *Post-commit*: Finally, a transaction changes its write intents to persistent writes in case of commit, and removes them in case of abort. This step is not essential for correctness, but reduces the overhead of future transactions. It occurs after the transaction is persistently committed or aborted via the commit entry, and can be done asynchronously.

2.4 Big data platforms

Apache HBase is one of the most scalable key-value storage technologies available today. Like many state-of-the-art data stores, it scales through horizontal *sharding* (partitioning) of data across *regions*. An HBase instance is deployed on multiple nodes (*region servers*), each of which typically serves hundreds of regions. Production HBase clusters of 1K nodes and above are becoming common. For example, Yahoo Japan leverages an HBase cluster of 3,800 nodes that collectively store 37PB of data [17].

Phoenix complements the HBase storage tier with a query processing (compute) tier. The latter scales independently (the current scalability goal is 10,000 query servers). Phoenix compiles every SQL statement into a plan, and executes it on one or more servers. Its query processing code invokes the underlying HBase for low-level data access, and a TPS (Omid or Tephra) for transaction management, through client libraries.

Wherever possible, Phoenix strives to push computation close to data (e.g., for filtering and aggregation), in order to minimize cross-tier communication. For this, it makes extensive use of server-side stored procedures, which in HBase are supported by the non-intrusive *coprocessor* mechanism. Omid uses HBase coprocessors too, both for performance improvements and for specific services, such as garbage collection of redundant data.

3. LOW-LATENCY TRANSACTIONS

We now describe Omid LL, a scalable low-latency TPS algorithm that satisfies standard (unrelaxed) SI semantics and is amenable to multi-tenancy. We saw above that, while many TPSs follow a similar schema, they make different design choices when implementing this schema. We overview our design choices in Section 3.1. We then proceed to give a detailed description of the protocol in Section 3.2.

3.1 Omid LL design choices

We now discuss our design choices, which are geared towards high performance without sacrificing cloud deployability. Table 1 compares them with choices made in other TPSs. We are not familiar with another TPS that makes the same design choices as Omid LL.

Table 1: Design choices in TPSs. **C** – centralized, **D** – distributed, **R** – replicated.

TPS	validation	commit entry updates	multi tenancy	read force abort
Percolator,Omid 2PC	**D**	**D**	no	yes
CockroachDB	**D**	**D**	yes	yes
Omid1, Tephra	**C**	**R**	yes	no
Omid	**C**	**C**	yes	no
Omid LL	**C**	**D**	**yes**	**yes**

Centralized validation. Omid LL adopts Omid's centralized conflict detection mechanism, which eliminates the need for locking objects in the data store, and is extremely scalable [40].

Other TPSs (like Percolator and CockroachDB [8]) instead use a distributed 2PC-like protocol that locks all written objects during validation (either at commit time or during the write). To this end, they use atomic check&mutate operations on the underlying data store. This slows down either commits (in case of commit-time validation) or transactional writes (in case of write-time validation), which takes a toll on long transactions, where validation time is substantial. To allow us to compare the two approaches, we also implement a 2PC-based version of Omid, Omid 2PC, which follows Percolator's design.

Distributed commit entry updates with multi-tenancy. The early generation of Omid [34] (referred to as Omid1) and Tephra replicate commit entries of pending transactions among all active clients, which consumes high bandwidth and does not scale. Omid instead uses a dedicated commit table, and has the centralized TM persist all commits to this table. Our experiments show that the centralized access to commit entries is Omid's main scalability bottleneck, and

while this bottleneck is mitigated via batching, this also increases latency. Omid chose this option as it was designed for high throughput.

Here, on the other hand, we target low latency. We therefore distribute the commit entry updates, and allow commit entries of different transactions to be updated in parallel by independent clients. We will see below that this modification reduces latency by up to an order of magnitude for small transactions.

We note that commit table updates are distributed also in CockroachDB and Percolator. The latter takes this approach one step further, and distributes not only the commit table updates but also the actual commit entries. There, commit entries reside in user data tables, where the first row written in a given transaction holds the commit entry for that transaction. The problem with this approach is that it assumes that all clients have permissions to access all tables. For example, a transaction attempting to read from table A may encounter a write intent produced by a transaction that accessed table B before table A, and will need to refer to that transaction's commit entry in table B in order to determine its status. This approach did not pose a problem in Percolator, which was designed for use in a single application, but is unacceptable in multi-tenant settings.

Unlike data tables, which are owned by individual applications that manage their permissions, the dedicated commit table is owned by the TPS; it is accessed exclusively by the TPS client library, which in turn is only invoked internally by the database engine, and not by application code.

The commit table is a highly contended resource. While it is not very big at any given time, it is accessed by every transaction that modifies data. In large clusters, the update rate may become huge (see Section 5). We therefore shard this table evenly across many nodes in order to spread the load.

Write intent resolution. As in other TPSs, reads resolve write intents via the commit entry. If the transaction status is committed, the commit time is checked, and, if smaller than or equal to ts_r, it is taken into account; if the transaction is aborted, the value is ignored. In case the transaction status is pending, Omid FP, like Percolator and CockroachDB, has the reader force the writing transaction to abort. This is done using an atomic check&mutate operation to set the status in the writing transaction's commit entry to aborted.

Omid and Tephra, on the other hand, do not need to force such aborts[2], because they ensure that if the read sees a write intent by an uncommitted transaction, the latter will not commit with an earlier timestamp than the read. Omid LL avoids such costly mechanisms by allowing reads to force aborts.

3.2 Omid LL algorithm

Like Omid, Omid LL uses a dedicated *commit table (CT)* for storing commit entries. A transaction is atomically committed by adding to the CT a commit entry mapping its ts_r to its ts_c. The post-commit phase then copies this information to a dedicated *commit* column in the data table, in order to spare future reading transactions the overhead of checking the commit table. Figure 2 shows the metadata of a transaction during its stages of execution.

[2]Omid's high availability mechanism may force such aborts in rare cases of TM failover.

Data Table

key	value	version	commit
k_1	a	3	nil

Commit Table

ts_r	commit

(a) Pending transaction

Data Table

key	value	version	commit
k_1	a	3	nil

Commit Table

ts_r	commit
3	7

(b) Committed transaction

Data Table

key	value	version	commit
k_1	a	3	7

Commit Table

ts_r	commit
3	7

(c) Post-committed transaction

Figure 2: Evolution of Omid LL metadata during a transaction. The transaction receives $ts_r = 3$ in the begin stage and uses this as the version number for tentative writes. It receives a commit timestamp $ts_c = 7$ when committing.

Algorithm 2 Omid LL's TM algorithm with HA support.

```
1:  procedure BEGIN
2:      checkRenew()
3:      return Clock.fetchAndIncrement()
4:  procedure COMMIT(txid, write-set)
5:      checkRenew()
6:      ts_c ← Clock.fetchAndIncrement()
7:      if conflictDetect(txid, write-set, ts_c) then
8:          return ts_c
9:      else
10:         return ABORT
11: procedure CHECKRENEW ▷ HA support; δ is the lease time
12:     if lease < now + 0.2δ then                    ▷ extend lease
13:         renew lease for δ time                    ▷ atomic operation
14:         if failed then halt
15:     if Clock = epoch then                         ▷ extend epoch
16:         epoch ← Clock + range
17:         if ¬CAS(maxTS, Clock, epoch) then halt
```

Whereas Omid's CT is updated by the TM, Omid LL distributes the CT updates amongst the clients. Its TM is thus a simplified version of Omid's TM, and appears in Algorithm 2. It has two roles: First, it allocates begin and commit timestamps by fetching-and-incrementing a monotonically increasing global clock. Second, upon commit, it calls the *conflictDetect* function, which performs validation (i.e., conflict detection) using an in-memory hash table by checking, for each key in the write-set, that its commit timestamp in the hash-table is smaller than the committing transaction's ts_r. If there are no conflicts, it updates the hash table with the write-set of the new transaction and its ts_c. The *checkRenew* procedure supports the TM's high availability, and is explained at the end of this section.

Client operations proceed as follows (cf. Algorithm 3):

Begin. The client sends a begin request to the TM, which assigns it a read timestamp ts_r.

Write. The client uses the data store's update API to add a tentative record to the data store, with the written key and value, version number ts_r, and nil in the commit column. It also tracks key in its local write-set.

Read. The algorithm traverses data records (using the data store's ds.get API) pertaining to the requested key with

Algorithm 3 Omid LL's client-side operations.

```
procedure BEGIN
    return TM.begin

procedure WRITE(ts_r, key, fields, values)
    track key in write-set
    add commit to fields and nil to values
    return ds.put(key, ts_r, fields, values)

procedure READ(ts_r, key)
    for rec ← ds.get(key, versions down from ts_r) do
        ▷ set ts_c to the commit timestamp associated with rec
        if rec.commit ≠nil then                    ▷ commit cell exists
            ts_c ← rec.commit
        else                                       ▷ commit cell is empty, check CT
            ts_c ← CT.get(rec.ts)
            if ts_c = nil then                     ▷ no CT entry, set status to ⊥
                ts_c ←CT.check&mutate(rec.ts, commit, nil, ⊥)
                if ts_c=nil then                   ▷ forced abort successful
                    ts_c ← ⊥
            if ts_c = ⊥ then
                ▷ check for race: commit before ⊥ entry created
                re-read rec from ds
                if rec.commit ≠nil then
                    ts_c ← rec.commit
                    CT.remove(rec.ts)
                else continue                      ▷ writing transaction aborted
        if ts_c < ts_r then
            return rec.value
    return nil                                     ▷ no returnable version found in loop

procedure COMMIT(ts_r, write-set)
    ts_c ← TM.commit(ts_r, write-set)              ▷ may return abort
    if ts_c ≠ abort then
        if CT.check&mutate(ts_r, commit, nil, ts_c) = ⊥ then
            ts_c ← abort
                                                   ▷ post-commit
    for all keys k ∈ write-set do
        if ts_c = abort then ds.remove(k, ts_r)
        else ds.put(k, ts_r, commit, ts_c)
    CT.remove(ts_r)                                ▷ garbage-collect CT entry
```

a version that does not exceed ts_r, latest to earliest, and returns the first value that is committed with a version smaller than or equal to ts_r. To this end, it needs to discover the commit timestamp, ts_c, associated with each data record it considers.

If the commit cell of the record is empty (commit=nil), then we do not know whether the transaction that wrote it is still pending or simply not post-committed. Since read cannot return without determining its final commit timestamp, it must forcefully abort it in case it is still pending, and otherwise discover its outcome. To this end, read first refers to the CT. If there is no CT entry associated with the transaction (ts_c=nil), read attempts to create an entry with $\perp$ as the commit timestamp. This is done using an atomic check&mutate operation, due to a possible race with a commit operation; if a competing commit attempt finds the $\perp$ entry, it aborts. Note that it is possible to wait a configurable amount of time before attempting to create a $\perp$ entry, in order to allow the transaction to complete without aborting it.

There is, however, a subtle race to consider in case the commit record is set to $\perp$ by a read operation. Consider a slow read operation that reads the data record rec when its writing transaction is still pending, and then stalls for a while. In the interim, the writing transaction successfully commits, updates the data record during post-commit, and finally garbage-collects its CT entry. At this point the slow reader wakes up and does not find the commit entry in the CT. It then creates a $\perp$ entry even though the transaction had successfully committed. To discover this race, we have a read operation that either creates or finds a $\perp$ entry in the CT re-read the data record, and if it does contain a commit entry, use it.

Commit. The client first sends a commit request to the TM. If the TM detects conflicts then the transaction aborts, and otherwise the TM provides the transaction with its commit timestamp ts_c. The client then proceeds to commit the transaction, provided that no read had forcefully aborted it. To ensure the latter, the client uses an atomic check&mutate to create the commit entry.

To avoid an extra read of the CT on every transactional read, once a transaction is committed, the post-commit stage writes the transaction's ts_c to the commit columns of all keys in the transaction's write-set. In case of an abort, it removes the pending records. Finally, it garbage-collects the transaction's CT entry.

3.2.1 *High availability*

The TM is implemented as a primary-backup process pair to ensure its high availability. The backup detects the primary's failure using timeouts. When it detects a failure, it immediately begins serving new transactions, without any recovery procedure. Because the backup may falsely suspect the primary because of unexpected network or processing delays we take precautions to avoid correctness violations in cases when both primaries are active at the same time.

To this end, we maintain two shared objects, managed in Apache Zookeeper and accessed infrequently. The first is *maxTS*, which is the maximum timestamp an active TM is allowed to return to its clients. An active (primary) TM periodically allocates a new *epoch* of timestamps that it may return to clients by atomically increasing maxTS using an atomic compare-and-swap (CAS) operation. Following failover, the new primary allocates a new epoch for itself, and thus all the timestamps it returns to clients exceed those returned by previous primary.

The second shared object is a locally-checkable *lease*, which is essentially a limited-time lock. The lease is renewed for some parameter δ time once 80% of this time elapses. As with locks, at most one TM may hold the lease at a given time. This ensures that no client will be able to commit a transaction in an old epoch after the new TM has started using a new one.

4. FAST-PATH TRANSACTIONS

The goal of our fast path is to forgo the overhead associated with communicating with the TM to begin and commit transactions. This is particularly important for short transactions, where the begin and commit overhead is not amortized across many operations. We therefore focus on single-key transactions.

To this end, we introduce in Section 4.1 a streamlined *fast path (FP)* API that jointly executes multiple API calls of the original TPS. We proceed, in Section 4.2, to explain a high-level general fast path algorithm for any system that follows the generic schema of Algorithm 1 above. Finally,

in Section 4.3, we describe our implementation of the fast path in Omid FP, and important practical optimizations we applied in this context.

4.1 API

For brevity, we refer to the TPS's API calls begin, read, write, and commit as b, r, w, and c respectively, and we combine them to allow fast processing. The basic FP transactions are singletons, i.e., transactions that perform a single read or write. These are supported by the APIs:

brc(key) – begins an FP transaction, reads key within it, and commits.

bwc(key,val) – begins an FP transaction, writes val into a new version of key that exceeds all existing ones, and commits.

We further support a fast path transaction consisting of a read and a dependent write, via a pair of API calls:

br(key) – begins an FP transaction and reads the latest version of key. Returns the read value along with a version ver.

wc(ver, key,val) – validates that key has not been written since the br call that returned ver, writes val into a new version of key, and commits.

Read-only transactions never abort, but bwc and wc may abort. If an FP transaction aborts, it can be retried either via the fast path again, or as a regular transaction.

4.2 Generic fast path algorithm

Algorithm 4 Generic support for FP transactions; each data store operation is executed atomically.

Client-side logic:

```
1: procedure BRC(key)
2:     rec ← ds.get(last committed record of key)   ▷ no preGet
3:     return rec.value
4: procedure BWC(key, value)
5:     return ds.putVersion(∞, key, value)
6: procedure BR(key)
7:     rec ← ds.get(last committed record of key)   ▷ no preGet
8:     return ⟨rec.version, rec.value⟩
9: procedure WC(ver, key, value)
10:    return ds.putVersion(ver, key, value)
```

Data store (stored procedure) logic:

```
11: procedure PUTVERSION(old, key, value)
                                  ▷ used by FP writes (bwc and wc)
12:    if key has no tentative version ∧
13:       last committed version of key ≤ old then
14:       ver ← F&I(key's maxVersion) +1
15:       ds.put(key, value, ver, ver)              ▷ no prePut
16:       return commit
17:    else
18:       return abort
19: procedure PREGET(ts_r, key)
          ▷ executed atomically with ds.get call, once per key
20:    bump(key's maxVersion, ts_r)
21: procedure PREPUT(key, val, ts_r, nil)
       ▷ checked atomically before put (by transactional write)
22:    if key has a committed version that exceeds ts_r then
23:       abort
24: procedure PREUPDATE(key, ts_r, commit, ts_c)
    ▷ executed atomically with ds.update call (by post-commit)
25:    bump(key's maxVersion, ts_c)
```

The generic fast path algorithm, which we present in Algorithm 4, consists of two parts: client side-logic, and logic running within the underlying data store. The latter is implemented as a stored procedure, Specifically, it supports a new flavor of put, *putVersion*, which is used by singleton writes, and it extends the put, get, and update APIs used by regular transactions with additional logic. The new logic is presented in the *preGet*, *prePut*, and *preUpdate* procedures, which are executed before, and atomically with, the corresponding data store calls.

Singleton reads (line 1) simply return the value associated with the latest committed version of the requested key they encounter. They ignore tentative versions, which may lead to missing the latest commit in case its post-commit did not complete, but is allowed by our semantics. FP reads can forgo the begin call since they do not need to obtain a snapshot time a priori. They can also forgo the commit call, since they perform a single read, and hence their 'snapshot' is trivially valid.

In case bwc encounters a tentative version, it does not try to resolve it, but rather simply aborts. This may cause false aborts in case the transaction that wrote the tentative version has committed and did not complete post-commit, as well as in the case that it will eventually abort. In general, this mechanism prioritizes regular transactions over FP ones. We choose this approach since the goal is to complete FP transactions quickly, and if an FP transaction cannot complete quickly, it might as well be retried as a regular one.

Such a singleton write has two additional concerns: (1) it needs to produce a new version number that exceeds all committed ones and is smaller than any commit timestamp that will be assigned to a regular transaction in the future. (2) It needs to make sure that conflicts with regular transactions are detected.

To handle these concerns, we maintain the timestamps as two-component structures, consisting of a global version and a locally advancing sequence number. In practice, we implement the two components in one long integer, with some number ℓ least significant bits reserved for sequence numbers assigned by FP writes (in our implementation, $\ell = 20$). The most significant bits represent the global version set by the TM. The latter increases the global clock by 2^ℓ upon every begin and commit request.

To support (1) a bwc transaction reads the object's latest committed version and increments it. The increment is done provided that it does not overflow the sequence number. In the rare case when the lower ℓ bits are all ones, the global clock must be incremented, and so the FP transaction aborts and is retried as a regular one.

It is important to note that the singleton write needs to *atomically* find the latest version and produce a new one that exceeds it, to make sure that no other transaction creates a newer version in the interim. This is done by a new *putVersion* function implemented in code that resides at the data store level. In Section 4.3 below, we explain how we implement such atomic conditional updates in an HBase coprocessor as part of Omid FP. The first parameter to putVersion is an upper bound on the key's current committed version; since a singleton write imposes no constraints on the object's current version, its upper bound is ∞.

Next, we address (2) – conflicts between FP and regular transactions. In case an ongoing regular transaction writes

to a key before bwc accesses it, bwc finds the tentative write and aborts.

It therefore remains to consider the case that a regular transaction T_1 writes to some key after FP transaction FP_1, but T_1 must abort because it reads the old version of the key before FP_1's update. This scenario is illustrated in Figure 3. Note that in this scenario it is not possible to move FP_1 to 'the past' because of the read.

Figure 3: Conflict between FP transaction FP_1 and regular transaction T_1.

In order for T_1 to detect this conflict, the version written by FP_1 has to exceed T_1's snapshot time, i.e., ts_r. To this end, we maintain a new field *maxVersion* for each key, which is at least as high as the key's latest committed version. The data store needs to support two atomic operations for updating maxVersion. The first is *fetch-and-increment, F&I*, which increments maxVersion and returns its old value; F&I throws an abort exception in case of wrap-around of the sequence number part of the version. The second operation, *bump*, takes a new version as a parameter and sets maxVersion to the maximum between this parameter and its old value.

Singleton writes increment the version using F&I (line 14), and the post-commit of transactional writes (line 24) bumps it to reflect the highest committed version. Every transactional read bumps the key's maxVersion to the reading transaction's ts_r (line 19); transactional writes (line 21) are modified to check for conflicts, namely, new committed versions exceeding their ts_r.

In the example of Figure 3, T_1's read bumps x's maxVersion to its ts_r, and so FP_1, which increments x's maxVersion, writes 1 with a version that exceeds ts_r. Thus, T_1's write detects the conflict on x.

Note that this modification of transactional writes incurs an extra cost on regular (non-FP) transactions, which we quantify empirically in Section 5.

The br and wc operations are similar to brc and bwc, respectively, except that wc uses the version read by br as its upper bound in order to detect conflicting writes that occur between its two calls.

4.3 Implementation and optimization

Associating a maxVersion field with each key is wasteful, both in terms of space, and in terms of the number of updates this field undergoes. Instead, when implementing support for Omid FP's fast path in HBase, we aggregate the maxVersions of many keys in a single variable, which we call the *Local Version Clock (LVC)*.

Our implementation uses one LVC in each region server. Using a shared LVC reduces the number of updates: a transactional read modifies the LVC only if its ts_r exceeds it. In particular, a transaction with multiple reads in the same region server needs to bump it only once.

We implement the two required atomic methods - F&I and bump on the LVC using atomic hardware operations (F&I and CAS, respectively). The HBase coprocessor mechanism enforces atomic execution of the stored code blocks by holding a lock on the affected key for the duration of the operation. Thus, putVersion executes as an atomic block, and calls the LVC's F&I method inside this block. Similarly, the calls to bump from preGet and preUpdate execute inside an atomic block with the ensuing get an update, respectively.

Note that although the coprocessor only holds a lock on the affected key, the joint update of the key and the LVC is consistent because it uses a form of two-phase locking: when the stored procedure begins, its locks the key, then the atomic access to the LVC effectively locks the LVC during its update; this is followed by an update of the key's record and the key's lock being released.

The LVC is kept in memory, and is not persisted. Migration of region control across region servers, which HBase performs to balance load and handle server crashes, must be handled by the LVC management. In both cases, we need to ensure that the monotonicity of the LVC associated with the region is preserved. To this end, when a region is started in a new server (following migration or recovery), we force the first operation accessing it in the new server to accesses the TM to increment the global clock, and then bump the local clock to the value of the global clock. Since the LVC value can never exceed the global clock's value, this bootstrapping procedure maintains its monotonicity.

5. EVALUATION

We describe our methodology and experiment setup in Section 5.1, and present our results in Section 5.2.

5.1 Methodology

5.1.1 Evaluated systems

We now evaluate our protocol improvements to Omid. We evaluate Omid LL as described in Section 3 and Omid FP as described in Section 4. Our baseline is the legacy Omid implementation [40]. We also compare the systems to Omid 2PC, which uses distributed conflict resolution. Omid 2PC defers writes to commit time (as Percolator does), whence it writes and validates keys using check&mutate. In Omid FP, single-key transactions use the FP API, whereas longer transactions use the standard API. The other systems use the standard API only.

Note that both CockroachDB and Percolator use 2PC-based conflict resolution, so our Omid 2PC mimics their protocols. Direct comparison with the actual systems is not feasible because Percolator is proprietary and CockroachDB supports SQL transactions over a multi-tier architecture using various components that are incompatible with HBase. We also do no compare Omid FP to Omid1 and Tephra, since both significantly outperforms Omid1 [40].

To reduce latency, we configure the TPSs to perform the post-commit phase asynchronously, by a separate thread, after the transaction completes.

Note that Translytics requires many short transactions and long scans. In this paper we improve the performance of short transactions, while incurring few aborts. The latter implies that scans do not spend much time forcing aborts, and therefore, their performance is the same as in the vanilla Omid.

5.1.2 Experiment setup

Our experiment testbed consists of nine 12-core Intel Xeon 5 machines with 46GB RAM and 4TB SSD storage, interconnected by 10G Ethernet. We allocate three of these to HBase nodes, one to the TM, one to emulate the client whose performance we measure, and four more to simulate background traffic as explained below. Each HBase node runs both an HBase region server and the underlying Hadoop File System (HDFS) server within 8GB JVM containers.

Note that the only scalability bottleneck in the tested systems is the centralized TM. HBase, on the other hand, can scale horizontally across thousands of nodes, each of which processes a small fraction of the total load. Since each node typically serves millions of keys, data access rates remain load-balanced across nodes even when access to individual keys is highly skewed. And since read/write requests are processed independently by each HBase node, their performance remains constant as the workload and system size are scaled by the same factor.

Thus, to understand the system's performance at scale, we can run transactions over a small HBase cluster with an appropriately-scaled load, but need to stress the TM as a bigger deployment would. We do this at a ratio of 3:1000; that is, we run transactions on a 3-node HBase cluster and load the TM with a begin/commit request rate that would arise in a 1000-node HBase cluster with the same per-node load. For example, to test the client's latency at 100K tps, we have the TM handle 100K tps, and have an HBase deployment of three nodes handle read/write requests of 300 tps. As explained above, the HBase latency at 100K tps with 1000 servers would be roughly the same as in this deployment.

We use four machines to generate the background load on the TM using a custom tool [40] that asynchronously posts begin and commit requests on the wire and collects the TM's responses. We note that although in this experiment the TM maintains a small number of client connections (serving many requests per connection), the number in a true 1000-node system still falls well within the OS limit, hence no real bottleneck is ignored.

We measure the end-to-end client-incurred latency on a single client node that generates transactions over the 3-server HBase cluster. Note that the client also generates begin/commit requests, which account for $\sim 0.3\%$ of the TM's load. The client runs the popular YCSB benchmark [25], exercising the synchronous transaction processing API in a varying number of threads.

5.1.3 Workload

Our test cluster stores approximately 23M keys (~7M keys per node). The values are 2K big, yielding roughly 46GB of actual data, replicated three-way in HDFS. The keys are hash-partitioned across the servers. The data accesses are 50% reads and 50% writes. The key access frequencies follow a Zipf distribution, generated following the description in [36], with $\theta = 0.8$, which yields the abort rate reported in the production delpoyment of Omid [40]. In this distribution, the first key is accessed 0.7% of the time.
We test the system with two transaction mixes:

Random mix – transaction sizes (number of reads and writes) follow a Zipf distribution with $\theta = 0.99$, with a cutoff at 10. With these parameters, 63% of the transactions access three keys or less, and only 3% access 10 keys. We vary the system load from 30K to 500K transactions per second (tps).

BRWC – 80% of the transactions are drawn from the random mix distribution, and 20% perform a read and then a write to the same key.

We add the BRWC workload since single-key read+write transactions are common in production, but are highly unlikely to occur in our random mix, which uses random key selection with billions of keys.

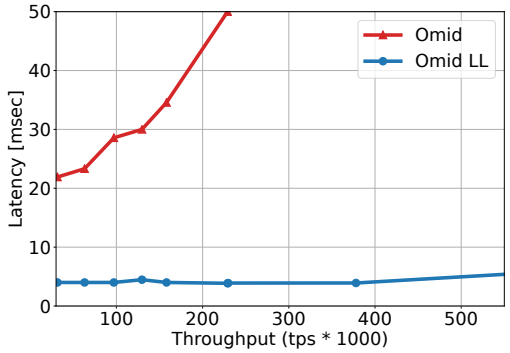


Figure 4: Throughput vs. latency, transaction size = 1.

5.2 Results

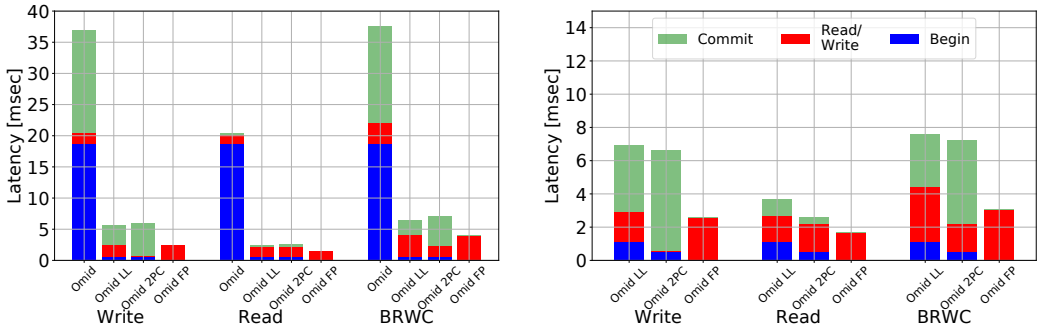
Throughput and latency of short transactions. Omid FP is motivated by the prevalence of short transactions in production, and is designed with the goal of accelerating such transactions. Its advantage is less pronounced for long transactions, where the cost of begin and commit is amortized across many reads and writes. To make the comparison meaningful, we classify transactions by their lengths and whether they access a single key, and study each transaction class separately.

We begin with short transactions. Figure 4 presents the average latency of single-key transactions run as part of the random mix, as a function of system throughput. Figure 5a then zooms in on the latency of such transactions under a throughput of 100K tps, and breaks up the different factors contributing to it. Figure 5b presents a similar breakdown under a high load of 500K tps; Omid is not included since it does not sustain such high throughput.

As we can see, under light load, Omid LL and Omid 2PC improve the latency of Omid by 4x to 5x. This is because in Omid, both begin and commit wait for preceding transactions to complete the writes of their commit entries; this stems from Omid's design choice to avoid the need for resolving pending write intents by aborting transactions; see penultimate column in Table 1. Single-key writes suffer from both the begin and commit latencies, whereas single-key reads suffer only from begins (Figure 5a). Omid 2PC has longer commit latencies and shorter write latencies because it defers writes to commit time.

As load increases, Omid suffers from a well-pronounced bottleneck, and its latency at 250K tps is doubled, where the other systems are unaffected. The extra delay in Omid is due to batching of commit record updates, which its TM applies to handle congestion [40].

Under low load, Omid LL is slightly faster than Omid 2PC (due to the latter's use of atomic check&mutate). But under high load, Omid 2PC is slightly better (10% faster on



(a) Low load (100K tps). **(b)** High load (500K tps).

Figure 5: Latency breakdown for single-key transactions under random mix workload.

average for single-key transactions) due to the centralized conflict analysis becoming a bottleneck in Omid LL.

The FP API delivers better performance for this traffic. For instance, under low load (Figure 5a), single writes take an average of 2.4ms using the bwc API versus 5.7ms in Omid LL (and 5.9ms in Omid 2PC). For comparison, a native HBase write takes roughly 2ms under this load. A single read executed using brc takes 1.5ms, which is the average latency of a native HBase read, versus 2.5ms as a regular transaction in Omid LL (2.7ms in Omid 2PC). For transactions that read and write a single key as part of the BRWC workload, the fast path implementation (consisting of br and wc calls) completes within 4ms, versus 6.5ms for Omid LL, 7.1ms for Omid 2PC, and 37.6ms for Omid. Under high load (Figure 5b), the fast path is even more beneficial: it reduces the latency of both read and write by more than a factor of 2.

Long transactions. We now examine longer transactions run as part of the random mix. Figure 6 shows the results for transactions of lengths 5 and 10. We see that the absolute latency gap of the new systems over Omid remains similar, but is amortized by other operations. Omid's control requests (begin and commit) continue to dominate the delay, and comprise 68% of the latency of 10-access transactions (under low load). In contrast, the transaction latency of Omid LL (and Omid FP) is dominated by data access, as only 13% (resp. 18%) of the time is spent on the control path. Omid 2PC spends 66% time executing commit because it defers writes to commit time, but this is offset by a shorter read/write execution time.

The FP mechanism takes a toll on the data path, which uses atomic check&mutate operations instead of simple writes. This is exacerbated for long transactions. For example, a 10-access transaction takes 24.8ms with Omid FP, versus 21.7ms with Omid LL. The performance of Omid 2PC with long transactions is similar to that of Omid FP – e.g., 25.6ms for 10-access transactions – because it also uses check&mutate operations during the commit validation phase.

Figure 7 summarizes the tradeoffs entailed by Omid FP relative to Omid LL for the different transaction classes. We see that under low load (Figure 7a), the speedup for single-write transactions is 2.3x, whereas the worst slowdown is 13%. In systems geared towards real-time processing, this is a reasonable tradeoff, since long transactions are infrequent and less sensitive to extra delay. Under high load (Figure 7b), the fast path is even more advantageous: the speedup for writes is 2.7x.

Abort rates. We note that Omid FP yields slightly higher rates of transaction aborts compared to Omid (recall that Omid LL aborts tentative writes in favor of concurrent reads, whereas Omid FP also aborts singleton writes in presence of concurrent tentative writes). However, the abort rates exhibited by all the systems are minor. Here, under the highest contention, Omid FP aborts approximately 0.1% of the transactions versus Omid LL's 0.08%, Omid 2PC's 0.08% and Omid's 0.07%.

6. SQL ORIENTED FEATURES

Comprehensive SQL support in Phoenix involved functionality extensions as well as performance optimizations.

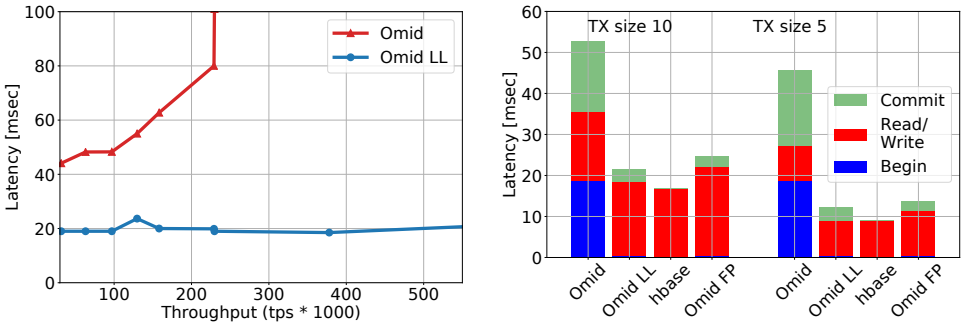
Performance-wise, Phoenix extensively employs stored procedures implemented as HBase coprocessors in order to eliminate the overhead of multiple round-trips to the data store. We integrated Omid's code within such HBase-resident procedures. For example, Phoenix coprocessors invoke transactional reads, and so we implemented Omid's transactional read – the loop implementing read in Algorithm 3 – as a coprocessor as well. This allowed for a smooth integration with Phoenix and also reduced the overhead of transactional reads when multiple versions are present.

Functionality-wise, we added support for on-the-fly construction of secondary indexes (see Section 6.1) and extended Omid's SI model to allow for multiple read- and write-points, as required in some Phoenix applications (see Section 6.2).

6.1 Index construction

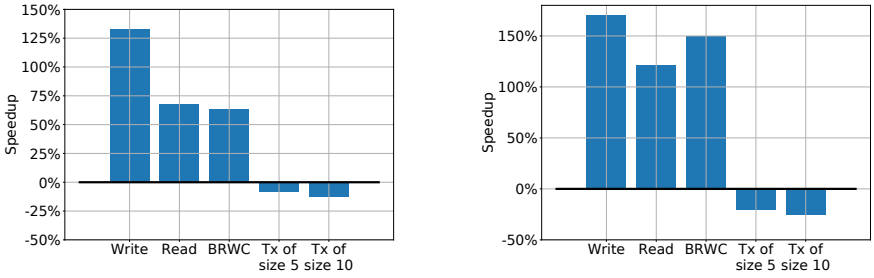
A secondary index in SQL is an auxiliary table that provides fast access to data in a table by a key that is different from the table's primary key. This need often emerges in analytics scenarios, in which data is accessed by multiple dimensions. Typically, the secondary key serves as the primary key of the secondary index, and is associated with a unique reference into the base table (e.g., primary key + timestamp). SQL query optimizers exploit secondary indexes in order to produce efficient query execution plans. Query speed is therefore at odds with update speed since every write to a table triggers writes to all its indexes.

The SQL standard allows creating indexes on demand. When a user issues a CREATE INDEX command, the database (1) populates the new index table with historic data from the base table, and (2) installs a coprocessor to augment every new write to the base table with a write to the index table. It is desirable to allow temporal overlap between the



(a) Throughput vs latency, transaction size=10 **(b)** Latency breakdown, transaction size = 5, 10

Figure 6: Latency vs. throughput and latency breakdown for long transactions in random mix workload.



(a) Low load (100 tps) **(b)** High load (500 tps)

Figure 7: Latency speedup with fast path API in Omid FP.

two, in order to avoid stalling writes while the index is being populated.

We exploit Omid's SI semantics in order to create such indexes. To achieve (1), Phoenix invokes a transaction that scans a snapshot of the base table and streams the data into the index table. This way, historic data is captured without blocking concurrent puts. Phoenix has two types of indexes: global (sharded independently of the base table) and local (co-sharded with the base table). In the latter case, the index building transaction is pushed into a coprocessor; this local distributed execution speeds up the process significantly.

Once the bulk population completes, the index can become available to queries. To achieve (2), the database creates a coprocessor that augments all transactions that update the base table with an additional update of the secondary index.

In order to guarantee the new index's consistency with respect to the base table, the snapshot creation and the trigger setup must be atomic. In other words, all writes beyond the snapshot timestamp must be handled by the coprocessor. Omid achieves this through a new *fence* API implemented by the TM, which is invoked when the coprocessor is installed. A fence call produces a new *fence timestamp* by fetching-and-incrementing the TM's clock, and records the fence timestamp in the TM's local state. Subsequently, the TM aborts every transaction whose read timestamp precedes the fence's timestamp and attempts to commit after it. Note that even though building an index can take hours, the fence operation is an atomic operation that occurs at index creation. Therefore, the fence only forces aborts of transactions that are active at the point in time when the fence is created and write to the fence's table. In practice, this amounts to around 10 transactions, and even if transactions are long, it is still less than 100 transactions. Moreover, there is at most one fence timestamp per index at a given time.

Neither the bulk index population nor its incremental update require write conflict detection among the index keys, for different reasons. The former does not contend with any other transaction, and hence is committed automatically – the commit cells are created simultaneously with the rest of the index data. The latter is voided by the TM detecting conflicts at the base table upon commit. Hence, in transactions augmented with secondary index updates, there is no need to embed the added index keys in the commit request, and so the load on the TM does not increase. Omid provides extensions for its put API for this scenario.

6.2 Extended snapshot semantics

Some Phoenix applications require semantics that deviate from the standard SI model in that a transaction does not see (all of) its own writes. We give two examples.

Checkpoints and snapshots. Consider a social networking application that stores its adjacency graph as a table of neighbor pairs. The transitive closure of this graph is computed in multiple iterations. Each iteration scans the table, computes new edges to add to the graph, and inserts them back into the table. Other functions like PageRank are implemented similarly. Such programs give rise to the pattern given in Figure 8.

Note that the SQL statement loops over entries of T. Normally, the SI model enforces read-your-own-writes semantics. However, in this case the desirable semantics are that the reads only see data that existed prior to the current statement's execution. This isolation level is called *snapshot isolation exclude-current (SIX)*.

To support this behavior, we implement in Omid two new methods, *snapshot* and *checkpoint*. Following a checkpoint, we move to the SIX level, where reads see the old snapshot

SQL program that requires SIX isolation:

```
for iterations i = 1,... do
    insert into T as select func(T.rec) from T where ...
```

Calls generated by Phoenix for the SQL program:

```
for iterations i = 1,... do
    checkpoint                                  ▷ move to SIX isolation
    iterator ← T.scan(...)                      ▷ scan in current snapshot
    for all rec ∈ iterator.getNext() do         ▷ parallel loop
        T.put(func(rec))                        ▷ write to next snapshot
    snapshot                                    ▷ make all previous writes visible
```

Figure 8: SQL program requiring SIX isolation and corresponding Phoenix execution plan.

(i.e., updates that precede the checkpoint call) and writes occur in the next snapshot, which is not yet visible. The snapshot call essentially resets the semantics back to SI, making all previous writes visible. Given these two calls, Phoenix translates SQL statements as above to the loop in Figure 8– it precedes the evaluation of the nested SQL statement with a checkpoint, and follows it with a snapshot. Thus, each SQL statement sees the updates of the preceding iteration but not of the current one.

To support the SIX isolation level in Omid, the TM promotes its transaction timestamp counter ts_r in leaps of some $\Delta > 1$ (rather than 1 as in the legacy system). The client manages two distinct local timestamps, τ_r for reads and τ_w for writes. By default, it uses $\tau_r = \tau_w = ts_r$, which achieves the traditional SI behavior. Omid then has the new methods, snapshot and checkpoint, increment τ_r and τ_w, respectively. If the consistency level is set to SIX, it maintains $\tau_w = \tau_r + 1 < ts_r + \Delta$, thereby separating the reads from the writes. A transaction can generate $\Delta - 1$ snapshots without compromising correctness. By default, Omid uses $\Delta = 50$.

Snapshot-all. When a transactions involves multiple snapshots that update the same key, it is sometimes required to read *all* versions of key that existed during the transaction. This is called the *snapshot-all* isolation level.

We use this isolation level, for example, when aborting a transaction that involves multiple updates to a key indexed by a secondary index. Consider a key k that is initially associated with a value v_0, and a secondary index maps v_0 to k. When a transaction updates k to take value v_1, a tentative deletion marker is added to v_0 in the secondary index while v_1 is added. If the transaction then updates the same key (in an ensuing snapshot) to take the value v_2, then v_1 is marked as deleted and v_2 is added, and so on. In case the transaction aborts, we need to roll-back all deletions. But recall that index updates are not tracked in the write-set, and so we need to query the data store in order to discover the set of updates to roll back. We do this by querying k in the primary table with the snapshot-all isolation level, to obtain all values that were associated with k throughout the transaction, beginning with v_0. We can then clean up the redundant entries in the secondary index, and remove the deletion marker from v_0.

7. RELATED WORK

The past few years have seen a growing interest in distributed transaction management [38, 27, 18, 20, 42, 33, 32]. Recently, many efforts have been dedicated to improving performance using advanced hardware trends like RDMA and HTM [43, 28, 29]. These efforts are, by and large, orthogonal to ours.

Our work follows the line of industrial systems, such as Google's Spanner [26], Megastore [19], and Percolator [39], CockroachDB [8], and most recently Apache Phoenix [6]; the latter employs transaction processing services provided by either Tephra [14] or Omid [40]. Production systems commonly develop transaction processing engines on top of existing persistent highly-available data stores: for example, Megastore is layered on top of Bigtable [24], Warp [32] uses HyperDex [31], and CockroachDB [8] uses RocksDB [13]. Similarly to Omid, Omid FP is layered atop HBase [2].

As discussed in Section 3, a number of these systems follow a common paradigm with different design choices, and we choose a new operation point in this design space. In particular, Omid LL eliminates the bottleneck of Omid by distributing the commit entry, makes commits and begins faster than Omid's by allowing reads to abort pending transactions. Unlike Percolator and CockroachDB, Omid LL uses scalable centralized conflict detection. This eliminates the need to modify HBase to support locking, and facilitates committing the algorithm back to the production system.

As a separate contribution we developed a fast path for single-key transactions, which is applicable to any of the aforementioned systems, and embodied it in Omid FP. A similar mechanism was previously developed in Mediator [23] for the earlier generation of Omid [34]. Mediator focused on reconciling conflicts between transactions and native atomic operations rather than adding an FP API as we do here. As a consequence, Mediator is less efficient than our fast path, and moreover, its regular transactions can starve in case of contention with native operations.

Our implementation of the fast path uses a local version clock. Similar mechanisms were used in Mediator and in CockroachDB, which uses per-region Hybrid Logical Clocks [37] for distributed timestamp allocation.

8. CONCLUSION

As transaction processing services begin to be used in new application domains, low transaction latency and rich SQL semantics become important considerations. In addition, public cloud deployments necessitate solutions compatible with multi-tenancy. Motivated by such use cases we evolved Omid for cloud use.

As part of this evolution, we improved Omid's protocol to reduce latency (by up to an order of magnitude) and improve throughput scalability. We further designed a generic *fast path* for single-key transactions, which executes them almost as fast as native HBase operations (in terms of both throughput and latency), while preserving transactional semantics relative to longer transactions. Our fast path algorithm is not Omid-specific, and can be similarly supported in other transaction management systems. Our implementation of the fast path in Omid FP can process single-key transactions at a virtually unbounded rate, and improves the latency of short transactions by another 3x–5x under high load.

We have extended Omid with functionalities required in SQL engines, namely secondary index construction and multi-snapshot semantics. We have successfully integrated it into the Apache Phoenix translytics engine.

Acknowledgments

We thank James Taylor for fruitful discussions.

9. ADDITIONAL AUTHORS

10. REFERENCES

[1] Amazon EMR. https://aws.amazon.com/emr/.

[2] Apache HBase. http://hbase.apache.org.

[3] The Apache Hive data warehouse. https://hive.apache.org.

[4] Apache Impala. https://impala.apache.org.

[5] Apache Omid. https://omid.incubator.apache.org/.

[6] Apache phoenix. https://phoenix.apache.org.

[7] Chatter. https://www.salesforce.com/eu/products/chatter/overview/.

[8] CockroachDB. https://github.com/cockroachdb/cockroach/blob/master/docs/design.md.

[9] Omid-82. https://issues.apache.org/jira/browse/OMID-82.

[10] Omid-90. https://issues.apache.org/jira/browse/OMID-90.

[11] Opentsdb – the scalable time series database. http://opentsdb.net.

[12] Phoenix-3623. https://issues.apache.org/jira/browse/PHOENIX-3623.

[13] RocksDB. http://rocksdb.org/.

[14] Tephra: Transactions for Apache HBase. https://tephra.io.

[15] The Forrester Wave: Translytical Data Platforms, Q4 2017. https://reprints.forrester.com/#/assets/2/364/RES134282/reports.

[16] Who is using Phoenix? https://phoenix.apache.org/who_is_using.html.

[17] Yahoo Japan: Hortonworks customer profile. https://hortonworks.com/customers/yahoo-japan-corporation/.

[18] Aguilera, M. K., Leners, J. B., and Walfish, M. Yesquel: Scalable sql storage for web applications. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), SOSP '15, pp. 245–262.

[19] Baker, J., Bond, C., Corbett, J. C., Furman, J., Khorlin, A., Larson, J., Leon, J.-M., Li, Y., Lloyd, A., and Yushprakh, V. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)* (2011), pp. 223–234.

[20] Balakrishnan, M., Malkhi, D., Wobber, T., Wu, M., Prabhakaran, V., Wei, M., Davis, J. D., Rao, S., Zou, T., and Zuck, A. Tango: Distributed data structures over a shared log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), SOSP '13, pp. 325–340.

[21] Berenson, H., Bernstein, P. A., Gray, J., Melton, J., O'Neil, E. J., and O'Neil, P. E. A critique of ANSI SQL isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995.* (1995), pp. 1–10.

[22] Borthakur, D., Gray, J., Sarma, J. S., Muthukkaruppan, K., Spiegelberg, N., Kuang, H., Ranganathan, K., Molkov, D., Menon, A., Rash, S., Schmidt, R., and Aiyer, A. Apache hadoop goes realtime at facebook. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2011), SIGMOD '11, ACM, pp. 1071–1080.

[23] Bortnikov, E., Hillel, E., and Sharov, A. Reconciling transactional and non-transactional operations in distributed key-value stores. In *Proceedings of International Conference on Systems and Storage* (New York, NY, USA, 2014), SYSTOR 2014, ACM, pp. 10:1–10:10.

[24] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst. 26*, 2 (June 2008), 4:1–4:26.

[25] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (New York, NY, USA, 2010), SoCC '10, ACM, pp. 143–154.

[26] Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., and Woodford, D. Spanner: Google's globally-distributed database. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)* (2012), pp. 261–264.

[27] Cowling, J., and Liskov, B. Granola: Low-overhead distributed transaction coordination. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (2012), USENIX ATC'12, pp. 21–21.

[28] Dragojević, A., Narayanan, D., Castro, M., and Hodson, O. Farm: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)* (Apr. 2014), pp. 401–414.

[29] Dragojević, A., Narayanan, D., Nightingale, E. B., Renzelmann, M., Shamis, A., Badam, A., and Castro, M. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), SOSP '15, pp. 54–70.

[30] Elnikety, S., Zwaenepoel, W., and Pedone, F. Database replication using generalized snapshot isolation. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS 2005),26-28 October 2005, Orlando, FL, USA* (2005), IEEE Computer Society, pp. 73–84.

[31] Escriva, R., Wong, B., and Sirer, E. G. Hyperdex: A distributed, searchable key-value store. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (2012), SIGCOMM '12, pp. 25–36.

[32] Escriva, R., Wong, B., and Sirer, E. G. Warp: Lightweight multi-key transactions for key-value stores. *CoRR abs/1509.07815* (2015).

[33] Eyal, I., Birman, K., Keidar, I., and van Renesse, R. Ordering transactions with prediction in distributed object stores. In *LADIS* (2013).

[34] Ferro, D. G., Junqueira, F., Kelly, I., Reed, B., and Yabandeh, M. Omid: Lock-free transactional support for distributed data stores. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014* (2014), pp. 676–687.

[35] Gray, J., and Reuter, A. *Transaction Processing: Concepts and Techniques*, 1st ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[36] Gray, J., Sundaresan, P., Englert, S., Baclawski, K., and Weinberger, P. J. Quickly generating billion-record synthetic databases. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 1994), SIGMOD '94, ACM, pp. 243–252.

[37] Kulkarni, S., Demirbas, M., Madeppa, D., Avva, B., and Leone, M. Logical physical clocks and consistent snapshots in globally distributed databases.

[38] Patterson, S., Elmore, A. J., Nawab, F., Agrawal, D., and Abbadi, A. E. Serializability, not serial: Concurrency control and availability in multi-datacenter datastores. In *PVLDB* (2012), vol. 5, pp. 1459–1470.

[39] Peng, D., and Dabek, F. Large-scale incremental processing using distributed transactions and notifications. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)* (2010).

[40] Shacham, O., Perez-Sorrosal, F., Bortnikov, E., Hillel, E., Keidar, I., Kelly, I., Morel, M., and Paranjpye, S. Omid, reloaded: Scalable and highly-available transaction processing. In *15th USENIX Conference on File and Storage Technologies (FAST)* (2017).

[41] Shute, J., Vingralek, R., Samwel, B., Handy, B., Whipkey, C., Rollins, E., Oancea, M., Littleeld, K., Menestrina, D., Ellner, S., Cieslewicz, J., Rae, I., Stancescu, T., and Apte, H. F1: A distributed sql database that scales. In *VLDB* (2013).

[42] Thomson, A., Diamond, T., Weng, S.-C., Ren, K., Shao, P., and Abadi, D. J. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (2012), SIGMOD '12, pp. 1–12.

[43] Wei, X., Shi, J., Chen, Y., Chen, R., and Chen, H. Fast in-memory transaction processing using rdma and htm. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), SOSP '15, pp. 87–104.