Low-Latency Transactions in Distributed Data Stores

Yonatan Gottesman* Aran Bergman† Edward Bortnikov*
Eshcar Hillel* Idit Keidar*† Ohad Shacham*
*Yahoo Research †Technion

Submission Type: Research

Abstract

We present Fragola, a highly scalable transaction processing engine for Apache HBase. The Fragola implementation is based on Apache Incubator Omid, but improves its protocol to reduce latency while at the same time improving throughput. Under light load, Fragola is 4x to 5x faster than Omid, and under high loads, it is more than an order of magnitude faster. Fragola further implements a *fast path* for single-key transactions, processing them almost as fast as native HBase operations, while preserving transactional semantics relative to longer transactions.

1 Introduction

In recent years, transaction processing [25] technologies have paved their way into many-petabyte big data platforms [29, 17, 30]. Modern industrial systems [29, 30, 8, 5] complement existing underlying key-value storage with *atomicity*, *consistency*, *isolation* and *durability* (ACID) semantics that enable programmers to perform complex data manipulation without over-complicating their applications. Transaction support in web-scale applications started from specific use cases like real-time content indexing [29, 30] but quickly expanded to full-scale SQL-compliant OLTP and online analytics [4, 31].

Similarly to many technologies, the adoption of transactions took a "functionality-first" trajectory. For example, the developers of Google Spanner [17] 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 high performance is rapidly picking up. Whereas early transaction systems were not latency-sensitive [29, 30], with the thrust into new interactive domains like messaging [13] and algorithmic trading [6], latency becomes essential. This paper is motivated by such applications.

Consider, for example, the platform powering Yahoo! Mail, which serves hundreds of millions of users. The Mail backend ingests billions of messages daily; messages undergo both machine classification (e.g., for spam filtering, thread detection, and "smart views" [1]), and user manipulation (e.g., starring, tagging, and moving between folders). Mail users browse their mail and search for content, issuing many billions of requests a day; they expect a consistent experience – e.g., messages do not disappear when moved between folders, starred content gets prioritized in search, folder counters are reliable, etc.

The Yahoo! Mail metadata platform is built on top of Apache HBase [2], which provides reliable and scalable key-value storage. The system's first generation was built without transaction support. This exposed developers to very complex programming scenarios to achieve ACID behavior (e.g., consistent folder listings, atomic updates of multiple counters). We have looked into building the next generation using Omid [3], an Apache Incubator project which provides a transaction API on top of HBase, and is already in use at large scale at Yahoo [30]. Unfortunately, while this makes programming much easier, it also jeopardizes the real-time latency SLA for interactive user experience. For example, simple updates and point queries must complete within single-digit milliseconds, whereas Omid, which was designed for throughput-oriented data pipelines [30], can induce latencies of tens to hundreds of milliseconds under high loads.

Motivated by this example, we have developed Fragola – a low-latency *and* high-throughput transaction processing system for HBase. Similarly to other modern transaction managers [29, 17, 30, 5], Fragola provides a variant of *snapshot isolation (SI)* [12], which scales better than traditional serializability implementations. Fragola is based on Omid [3], but dissipates the principal bottleneck present therein. Its advantage is maximized for short transactions, which are prevalent in latency-sensitive applications. Fragola processes such transactions in a handful of milliseconds. The new protocol also doubles the system throughput.

As a separate contribution, we introduce a novel *fast path* algorithm for short single-key transactions that eliminates the begin/commit overhead entirely, and executes short transactions almost as fast as native HBase operations. This entails minor extensions to the underlying data

store. The fast path is orthogonal to other protocol aspects, and can be supported in other transaction processing services.

We have implemented Fragola[1] based on the open source Omid code[2], and extended the HBase code to enable fast path transactions[3]. Our experiments on mid-range hardware show substantial performance improvements. Under low load, even without the fast path, Fragola transactions are 4x to 5x faster than Omid's, and the fast path further reduces the latency of short transactions by another 55% on average. As system load increases, Omid's latency surges (at ∼150K tps in our tests), whereas Fragola's remains stable until we generate a higher load (∼250K tps), and increases four-fold at 500K tps. Fast path transactions incur no scalability bottlenecks, and continue to execute at the low latency of native HBase operations regardless of system load (thanks to HBase's near perfect horizontal scalability). This comes at the cost of a minor (less than 15%) negative impact on longer transactions. Additionally, Fragola has negligible impact on transaction abort rates.

The remainder of this paper is organized as follows: In Section 2 we define the API and semantics of a transaction processing service. Section 3 presents Fragola, without the fast path, and Section 4 then describes our support for fast path transactions. Section 5 presents an empirical evaluation. We review related work in Section 6 and conclude with Section 7.

2 Service API and Semantics

Fragola is a *Transaction Processing System (TPS)*, namely a service that runs atop an underlying data store and allows users to bundle multiple data store operations into a single atomic transaction. We describe the data model and API of the underlying data store in Section 2.1, and then proceed to define the transaction semantics provided by Fragola in Section 2.2.

2.1 Data store

The data store holds *objects* (often referred to as *rows*) 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 in the data store. We further assume that a write operation can specify the version number it writes to. The data store provides the following API:

get – returns the value with the highest version associated with a given key along with its version number.

The API further allows traversing (reading) earlier versions of the same key in descending order.

update – creates or updates an object (either the entire object or some of its fields) with a given key and version: if the version already exists, its value is updated; otherwise, a new version is added.

remove – removes an object with a given key and version.

check&mutate – data stores often provide means to atomically read and update a single object, e.g., HBase exports check&mutate operations, which are internally implemented using a per-row RW lock.

A separate process performs garbage collection of obsolete versions.

2.2 Transaction semantics

TPSs provide *begin* and *commit* APIs for delineating transactions: a *transaction* the sequence of read and write operations on different objects that occur between begin and commit. 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 snapshot isolation [12], which is provided by popular database technologies such as Oracle, PostgreSQL, and SQL Server, and TPSs such as Percolator, Omid, Tephra, and CockroachDB.

SI enforces a total order on committed transactions according to their commit times so that

1. each transaction's read operations see a consistent snapshot of the database reflecting write operations by exactly those transactions that committed prior to the transaction's start time; and
2. a transaction commits only if none of the items it updates has been modified since that snapshot.

Thus, under 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.

[1] https://github.com/yonigottesman/incubator-omid/tree/localTransactions

[2] https://omid.incubator.apache.org

[3] https://github.com/yonigottesman/hbase_local_transactions/tree/0.98-add-rmw

Algorithm 1 TPS operation schema.

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

Following a commit call, the transaction may successfully *commit*, whereby all of its operations take effect; in case of conflicts, (i.e., when two concurrent transactions attempt to update the same item), the transaction may *abort*, in which case none of its changes take effect.

3 Fast Transactions with Fragola

We now describe Fragola, a scalable low-latency TPS. Fragola is an evolution of the open-source Omid TPS, redesigned to reduce latency. We begin in Section 3.1 with some background on the modus operandi of existing TPSs that support SI, including Omid. We will see that, while many TPSs follow a similar schema, they make different design choices when implementing this schema. We discuss the design choices of Fragola in Section 3.2. We then proceed to give a detailed description of the Fragola protocol in Section 3.3.

3.1 Background: Schema of TPS operation

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)* [29, 24, 30, 8], 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 [29, 24, 30, 8].

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 [29].

In some solutions writes check for conflicts before declaring their intents [5], whereas in others, all conflict detection is deferred to commit time [29, 24, 30, 8].

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 14 of Algorithm 1 as we explain below, and is checked in order to resolve write intents in line 9. In some cases [29, 5], when the writing transaction status 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., [29, 24, 30, 8], 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 [29]. Validation can be centralized [24, 30, 8] or distributed [29, 5].

3. *Commit* or abort in one irrevocable atomic step. This is achieved by persistently writing to the *commit en-*

try, which can reside in a global table [30, 5] or alongside the first key written by the transaction [29].

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.

3.2 Fragola design choices

We now discuss our design choices, which are summarized and compared with other TPSs in Table 1.

Centralized validation. Percolator, the first TPS in this vein, used a distributed commit protocol that locked all written objects during validation. Unfortunately, while an object is locked, other transactions attempting to access it are blocked. Since clients may stall or fail while holding a lock, Percolator detects failures of unresponsive clients and aborts transactions on their behalf. To eliminate the need for such failure detection and recovery, newer systems prefer to rely on a centralized TM for validation [24, 30, 8].

CockroachDB [5] takes a different approach: it performs distributed validation at write time, by replacing writes with atomic validate-and-write operations that can abort either the current transaction or a conflicting one. Since CockroachDB's approach slows down transactional writes while Omid's centralized conflict detection is extremely scalable [30], Fragola adopts the centralized validation mechanism from Omid.

Distributed commit entry. The early generation of Omid [24] (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 and CockroachDB instead use dedicated tables. CockroachDB updates the table in a distributed manner, while Omid has the centralized TM persist all commits. Our experiments show that the centralized access to commit entries is Omid's main scalability bottleneck, and while this bottleneck is mitigated via batching commit table updates, 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 table: we store each transaction's commit entry alongside its first written key, which we call the *leader*, and store pointers to this leader at all other keys written by the transaction. Thus, transaction entries can be updated in parallel by independent clients, and such updates are no longer a performance bottleneck. We will see below that this modification reduces latency by up to an order of magnitude on small transactions.

key	value	version	commit	leader
k1	a	3	nil	⟨k1,3⟩
k2	b	3	nil	⟨k1,3⟩
k3	c	3	nil	⟨k1,3⟩

(a) Pending transaction

key	value	version	commit	leader
k1	a	3	7	⟨k1,3⟩
k2	b	3	nil	⟨k1,3⟩
k3	c	3	nil	⟨k1,3⟩

(b) Committed transaction

key	value	version	commit	leader
k1	a	3	7	⟨k1,3⟩
k2	b	3	7	⟨k1,3⟩
k3	c	3	7	⟨k1,3⟩

(c) Post-committed transaction

Figure 1: Evolution of Fragola metadata during a transaction. The transaction receives $ts_r = 3$ in the begin stage, and $ts_c = 7$ when committing; its leader is ⟨k1,3⟩.

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, Fragola, 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[4], 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. This is achieved either by delaying begin and commit requests until all transactions that concurrently attempt to commit complete (as in Omid), or sending information about all these transactions to the beginning client (as in Omid1 and Tephra). Fragola avoids such costly mechanisms by allowing reads to force aborts.

3.3 Fragola algorithm

The metadata used by Fragola for transaction management is stored in two additional columns associated with

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

TPS	validation scheme	validation time	commit entry location	commit entry update	resolving write intents on read may cause abort
Percolator	**D**	commit	**D**	**D**	yes
Omid1, Tephra	**C**	commit	**R**	**C**	no
Omid	**C**	commit	**C**	**C**	no
CockroachDB	**D**	write	**C**	**D**	yes
Fragola	**C**	commit	**D**	**D**	yes

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

Algorithm 2 Fragola'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
12:     if lease < now + δ′ 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
```

each object version in the data store: (1) *commit* holds the status of the transaction that wrote this version – *nil* if it is pending, its ts_c if committed, and otherwise *abort*; and (2) *leader* is a pointer to the writing transaction's commit entry, namely, the *commit* column of the first key written by the same transaction.

Since the commit needs to be an atomic step, the first key in the transaction's write-set is chosen to be the leader, and the transaction commits via a single atomic write of ts_c to the leader's commit column. Transactional read operations that find a nil commit entry refer to the leader to find out whether the value they are attempting to read has been committed. Figure 1 shows the metadata of a transaction during its stages of execution.

Fragola's TM is 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, it performs validation (i.e., conflict detection) upon commit using an in-memory hash table, and updates the table with the write-set of the new transaction and its ts_c. Since the conflict detection algorithm has been reported elsewhere [30], and the Fragola client need not be aware of its implementation details, we refrain from repeating the details of the *conflictDetect* function here. The *checkRenew* procedure supports the TM's high availability, and is explained in the next section.

Algorithm 3 Fragola's read and commit operations.

```
procedure READ(key)
    for rec ← ds.get(key, versions down from ts_r) do
        if rec.commit ∉ {nil, abort} then          ▷ committed
            if rec.commit < ts_r then
                return rec.value
        else if rec.commit = nil then
            ts_c ← CHECKLEADER(rec.leader)
            rec.commit ← ts_c                      ▷ helping
            if ts_c ≠ abort ∧ ts_c < ts_r then
                return rec.value
    return nil

procedure CHECKLEADER(⟨k, ts⟩)
    while true do
        leader ← ds.get(⟨k, ts⟩)
        if leader = nil then return abort          ▷ leader removed
        if leader.commit = nil then                ▷ abort transaction
            ds.check&mutate(leader.commit, nil, abort)
        else return leader.commit                  ▷ transaction is complete

procedure COMMIT(ts_r, write-set)
    ts_c ← TM.commit(ts_r, write-set)              ▷ may return abort
    if ts_c ≠ abort then
        leader ← getLeader(write-set)
        ok ← ds.check&mutate(leader.commit, nil, ts_c)
        if ¬ok 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.update(⟨⟨k, ts_r⟩, commit⟩, ts_c)
```

Algorithm 3 describes Fragola's implementation of read and commit operations. The client's operations proceed as follows:

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

Write. The client adds a tentative record to the data store, with the written key and value, version number ts_r, nil in the commit column, and a pointer to the first key written during the transaction in the leader 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 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. Upon encountering a tentative record (with commit=nil), the algorithm calls the CHECKLEADER function, which gets the leader's ts_c from the data store, and again returns the key if that ts_c is smaller than ts_r.

If the leader's commit entry is still nil, read cannot return without determining the final commit timestamp of the pending transaction. To this end, CHECKLEADER attempts to forcefully abort the pending transaction. This is done using the check&mutate operation, which atomically reads and sets the commit entry to abort in case its value is still nil. If check&mutate fails, this means that the transaction is no longer pending, and the next iteration in the loop re-reads the leader's entry. Note that during the second iteration, the record can no longer be nil, and so it always returns without attempting another check&mutate.

Commit. The client sends a commit request to the TM, which assigns it a commit timestamp ts_c and checks for conflicts. If the TM does not return abort, this means that the validation has been successful, and the client attempts to commit the transaction. Note that the transaction can commit only if no read has invalidated it. Therefore, the client uses the data store's check&mutate function to write ts_c to the leader's commit column only if the commit status is still nil.

To avoid an extra read of the leader 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.

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 (e.g., garbage collection stalls), we take precautions to avoid correctness violations in (rare) 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 a 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 is a locally-checkable *lease*, which is essentially a limited-time lock. As with locks, at most one TM may hold the lease at a given time (this requires the TMs' clocks to advance roughly at the same rate). 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 Fragola, and important practical optimizations we applied in this context.

4.1 API and semantics

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.

We note that though the FP API richness can potentially complicate development, this drawback may be abstracted away by high-level access semantics. For example, a SQL database (e.g., Phoenix [4]) can select the most appropriate API in its query optimizer, transparently to the user.

Semantics. With FP transactions, Fragola continues to establish a total order on all update transactions, both fast and regular ones, and transactions continue to observe snapshots that reflect a prefix of this total order. However, we relax the real-time order guarantee of SI for FP transactions relative to regular ones. For example, an FP transaction may return an older value than the latest one committed by a regular transaction, essentially serializing the FP transaction 'in the past'. Similarly, a regular transaction overlapping two FP ones may observe an update of the second and miss an update by the first, as illustrated in Figure 2, where FP2 is ordered 'in the past'. Yet we do enforce real-time order on regular transactions as well as on all updates of the same key.

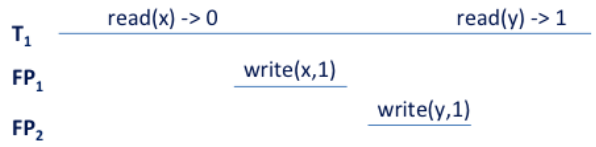


Figure 2: Possible violation of real-time order among fast path (FP) transactions. Regular transaction T_1 reads x before it is updated by FP transaction FP_1 and reads y after it is updated by FP transaction FP_2 even though FP_2 occurs after FP_1.

Formally, the system enforces a total order $\mathcal{T}$ on all committed transactions, so that

1. regular transactions (though not FP ones) are ordered in $\mathcal{T}$ according to their commit times;
2. non-overlapping transactions (regular and FP) that update the same key are ordered in $\mathcal{T}$ according to their commit times;
3. each regular transaction's read operations see a consistent snapshot of the database reflecting a prefix of $\mathcal{T}$ that includes at least all regular transactions committed prior to its start time; and
4. a transaction commits only if none of the items it updates is modified by a transaction ordered in $\mathcal{T}$ after its snapshot time $\mathcal{T}$.

Note that $\mathcal{T}$ preserves causality because only transactions that access different data objects can be re-ordered.

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)
3:      return rec.value
4:  procedure BWC(key, value)
5:      return ds.writeVersion(∞, key, value)
6:  procedure BR(key)
7:      rec ← ds.get(last committed record of key)
8:      return ⟨rec.version, rec.value⟩
9:  procedure WC(ver, key, value)
10:     return ds.writeVersion(ver, key, value)
```

Data store logic:

```
11: procedure WRITEVERSION(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:         add ⟨key, ver⟩ with commit = ver
16:         return commit
17:     else
18:         return abort
19: procedure TXREAD(ts_r, key)
                    ▷ used by transactional read, once per key
20:     bump(key's maxVersion, ts_r)
21:     return get(key)
22: procedure TXWRITE(ts_r, key)
                                  ▷ used by transactional write
23:     if key has a committed version that exceeds ts_r then
24:         abort
25:     add ⟨key, ts_r⟩ with commit = nil
26: procedure TXUPDATE(key, ver, ts_c)
                              ▷ used by transactional post-commit
27:     bump(key's maxVersion, ts_c)
28:     update commit field of ⟨key, ver⟩ to ts_c
```

The generic fast path algorithm consists of two parts: client side-logic, and new atomic operations implemented in the underlying data store. It is presented in pseudocode in Algorithm 4.

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 *writeVersion* function implemented at the data store level. In Section 4.3 below, we explain how we implement such atomic conditional updates in HBase as part of Fragola. The first parameter to writeVersion 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. (Note that the atomic conditional write in bwc takes care of conflicts among singletons.) 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 26) 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 22) 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 Fragola's fast path in HBase, we aggregate the maxVersions of many keys in a single variable, which we call the *Local Version Clock (LVC)*.

Like many state-of-the-art data stores, HBase data is *sharded* (partitioned) across many *regions*, and is deployed on multiple *region servers*, each of which serves several hundreds of regions. Our solution implements 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). In addition, HBase natively supports a check&mutate operation, which holds a lock on the affected key for the duration of the operation. Our implementation of the singleton write uses check&mutate to implement the atomic block, and calls the LVC's F&I method inside this block. Transactional reads and post-commits call bump from inside a check&mutate block where they do their read and version update, respectively.

Note that although check&mutate 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 check&mutate 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 bumps 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

Evaluated systems. We compare Fragola to a state-of-the art TPS and separately evaluate the effectiveness of it FP mechanism. To this end, we compare three TPSs, all of which use HBase as the underlying data store:

Vanilla Fragola – the algorithm described in Section 3, without support for FP transactions.

FP Fragola – the FP API implementation as described in Section 4, with transactional reads and writes modified to access the LVC along with the data.

Omid – the Apache Incubator version of Omid, on which we base our implementation of Fragola.

Direct comparison to other TPSs, e.g., Percolator or CockroachDB, 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 do no compare Fragola to Omid1 (and the similarly-designed Tephra), since Omid significantly outperforms Omid1 [30].

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

In Omid and Vanilla Fragola, all transactions use the standard API, namely delineating transactions with begin and commit calls. In FP Fragola, transactions that can use the FP API (specifically, singleton reads, singleton writes, and single-key read-writes) do so, whereas other transactions use the standard API.

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.

To test scalability, we project the load on the TM when deployed in a much larger system. Specifically, we project the TPSs' behavior when running transactions over a 1000-node HBase cluster. Such a cluster serves billions of keys, and data is sharded so that each region server holds roughly 0.1% of the data. Since in this setting each node serves millions of different keys, data access rates remain load-balanced across nodes even when access to individual keys is highly skewed. Since read/write requests are processed independently by each node, their performance remains constant as we scale the workload and the system size by the same factor. Hence, we can simulate transaction processing latency in a 1000-node cluster by using a small HBase cluster that serves a fraction of the read/write workload, while stressing the TM with the full load of begin and commit requests.

We emulate a 1000-node HBase instance using a 3-server cluster that processes 0.3% of the projected workload. The HBase traffic (on the three nodes) is driven by a client node running the popular YCSB benchmark [16], exercising the traditional (synchronous) transaction processing API in a varying number of threads. YCSB measures the end-to-end latencies incurred by the client. The remainder of the control requests (background load on the TM) are generated by a custom tool [30] that asynchronously posts them on the wire and collects the TM's responses. We use up to four machines to drive this traffic. Note that in this emulation the TM maintains a small

number of client connections (serving many requests per connection); however, the real number (in the emulated system) falls well within the OS limit, hence no real bottleneck is ignored.

Workload. Our test cluster stores approximately 23M keys (∼7M keys per node), which represent 7.66B keys in the emulated system. 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 [26], with $\theta = 0.8$, which we derive from production workloads in Yahoo's deployment of Omid [30]. Note that under this access distribution, data requests are load-balanced across nodes.

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.

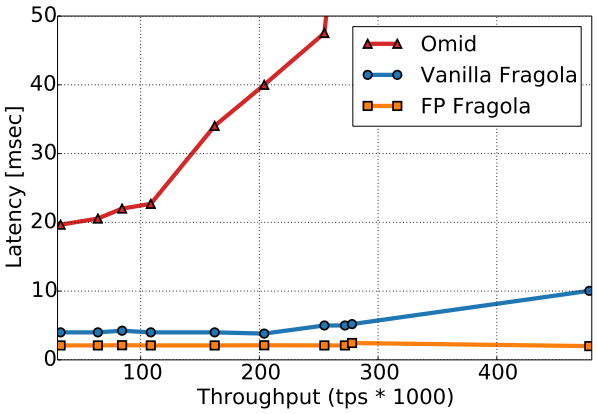


Figure 4: Throughput vs. latency, transaction size = 1.

5.2 Results

Throughput and latency of short transactions. Recall that Fragola is motivated by the prevalence of short transactions in production, and is designed with the goal of speeding such transactions up. 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.

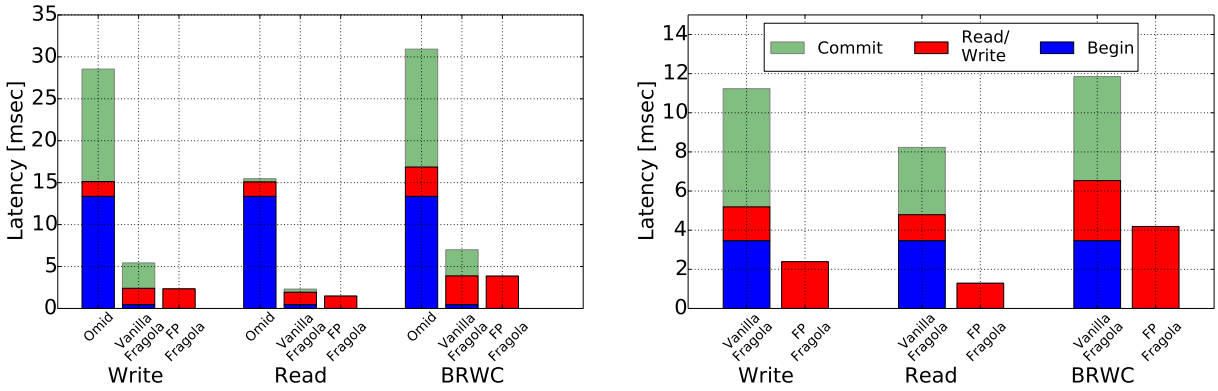
We begin with short transactions. Figure 4 presents the average latency of single-key transactions run as part of the random mix, as a function of system throughput. Figure 5a then zooms in on the latency of such transactions under a throughput of 100K tps, and breaks up the different factors contributing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

As we can see, under light load, Fragola improves the latency of Omid by 4x to 5x, even without the fast path. 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 last (rightmost) 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).

As load increases, Omid suffers from a well-pronounced bottleneck, and its latency at 200K tps is doubled, where Fragola's latency is unaffected. The extra delay in Omid is due to batching of commit record updates, which its TM applies to handle congestion [30]. Fragola, in turn, begins to experience a bottleneck (due to a heavy request load on the TM) at 250K tps. Such congestion arises due to Omid's centralized commit entry management (penultimate column in Table 1).

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 5.4ms using the regular transaction API (consisting of begin, write, and commit). 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 2.3ms as a regular transaction. 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 3.9ms, versus 7ms for Vanilla Fragola and 30.9ms for Omid. Under high load (Figure 5b), the fast path is more beneficial: it reduces the latency of write by nearly a factor of five, from 11.24ms to 2.4ms, and the read latency by a factor of six, from 8.24ms to 1.29ms.

Long transactions. We now examine longer transactions run as part of the random mix workload. Figure 6 shows the results for transactions of lengths 5 and 10. We



(a) Low load (100K tps).

(b) High load (500K tps).

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

see that the latency gap of Fragola over Omid remains similar, but is amortized by other operations. Omid's control requests (begin and commit) continue to dominate the delay, and comprise 61% of the latency of 10-access transactions. In contrast, Fragola's transaction latency is dominated by data access. For example, in 10-operation transactions only 17% of the time is spent on the control path, which leads to much faster completion.

Nevertheless, the FP mechanism takes its toll on the data path, which resorts to atomic check&mutate operations instead of simple writes. This is exacerbated for long transactions. For example, a 10-access transaction takes 24.4ms with FP Fragola, versus 20.8ms with Vanilla Fragola.

Figure 7 summarizes the tradeoffs entailed by the fast path API (relative to Vanilla Fragola) 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 14.8%. 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 clearly advantageous, most notably, the speedup for reads exceeds 500%.

Abort rates. We note that Fragola yields slightly higher rates of transaction aborts compared to Omid (recall that Vanilla Fragola aborts tentative writes in favor of concurrent reads, whereas FP Fragola 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, FP Fragola aborts approximately 0.1% of the transactions vs Vanilla Fragola's 0.08% and Omid's 0.07% (the latter is in line with [30]).

6 Related Work

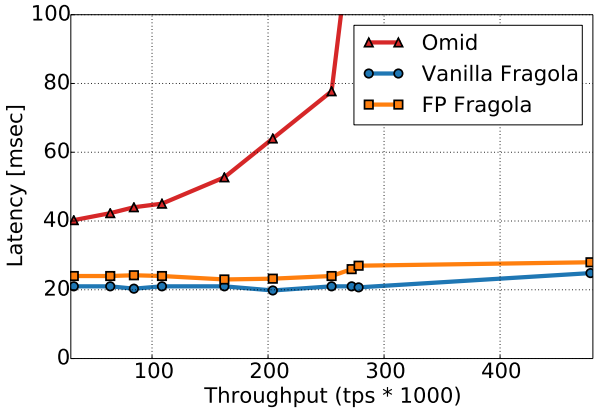
The past few years have seen a growing interest in distributed transaction management [28, 18, 9, 11, 32, 23, 22]. Recently, many efforts have been dedicated to improving performance using advanced hardware trends like RDMA and HTM [33, 19, 20]. These efforts are, by and large, orthogonal to ours.

Our work follows the line of industrial systems, such as Google's Spanner [17], Megastore [10], and Percolator [29], CockroachDB [5], and most recently Apache Phoenix [4]; the latter employs transaction processing services provided by either Tephra [8] or Omid [30][5]. 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 [15], Warp [22] uses HyperDex [21], and CockroachDB [5] uses RocksDB [7]. Similarly to Omid, Fragola is layered atop Apache HBase [2].

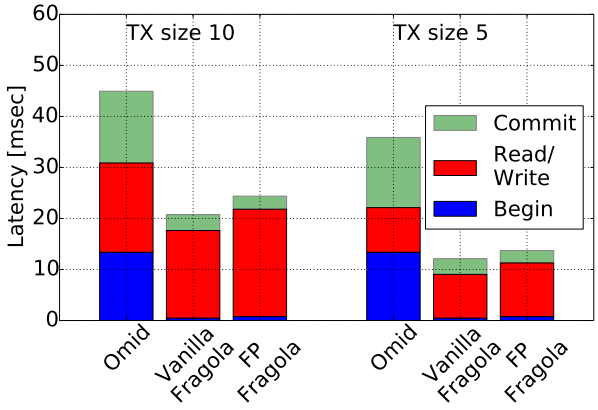
As discussed in Section 3, a number of these systems follow a common paradigm with different design choices, and Fragola chooses a new operation point in this design space. In particular, Fragola eliminates the bottleneck of Omid by distributing the commit entry, makes commits and begins faster than Omid's by allowing reads to abort pending transactions, but unlike Percolator and CockroachDB, uses centralized conflict detection. This eliminates the need to detect client failures as in Percolator or have transactional writes perform conflict detection as in CockroachDB.

As a separate contribution we developed a fast path for single-key transactions, which is applicable to any of the aforementioned systems. A similar mechanism was previously developed in Mediator [14] for the earlier gen-

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

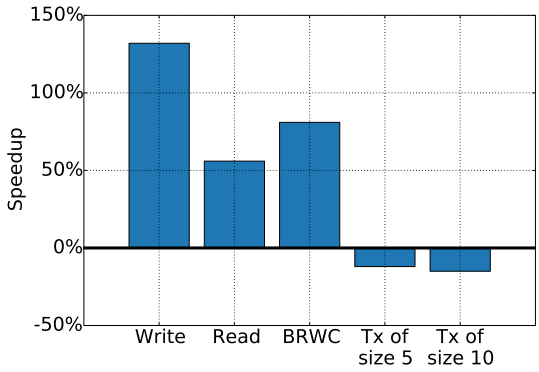


(a) Throughput versus latency, transaction size = 10

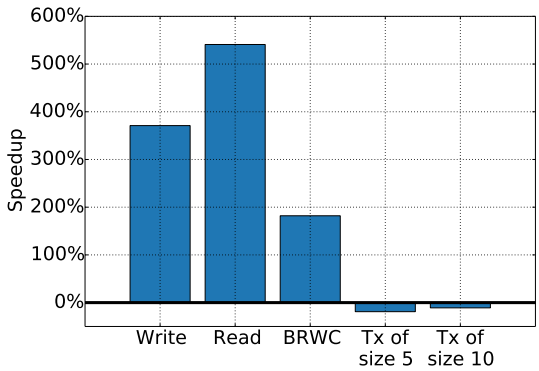


(b) Latency breakdown, transaction size = 5, 10

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



(a) Low load (100K tps)



(b) High load (500K tps)

Figure 7: Latency speedup with fast path API in Fragola.

eration of Omid [24]. 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 [27] for distributed timestamp allocation.

7 Conclusion

As transaction processing services begin to be used in new application domains, low transaction latency becomes an important consideration. Motivated by such use cases we developed Fragola, a highly scalable low-latency transaction processing engine for Apache HBase. We implemented Fragola based on Omid, and improved its protocol to reduce latency (by 4x to 5x under light load and an order of magnitude under heavy load), while also improving the throughput twofold.

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 Fragola-specific, and can be similarly supported in other transaction management systems. Our implementation of the fast path in Fragola can process single-key transactions at a virtually unbounded rate (thanks to HBase's horizontal scalability), and improves the latency of Fragola's short transactions by another 3x–5x under high load.

The Fragola code is publicly available[6], and we are working to contribute it back to Omid. We believe our improvements will become instrumental for modern cloud-based, large-scale, latency-sensitive services, e.g., the Phoenix OLTP system, which is designed to harness up to ten thousand nodes.

[6] https://github.com/yonigottesman/incubator-omid/tree/localTransactions

References

[1] https://yahoohelpcommunity.tumblr.com/post/118485031125/.

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

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

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

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

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

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

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

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

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

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

[12] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O'Neil, and P. E. O'Neil. 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.*, pages 1–10, 1995.

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

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

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

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

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

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

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

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

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

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

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

[24] D. G. Ferro, F. Junqueira, I. Kelly, B. Reed, and M. Yabandeh. 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*, pages 676–687, 2014.

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

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

[27] S. Kulkarni, M. Demirbas, D. Madeppa, B. Avva, and M. Leone. Logical physical clocks and consistent snapshots in globally distributed databases. 2014.

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

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

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

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

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

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