Fast Local Transactions in Distributed Data Stores

Paper ID

Number of pages

Abstract

TBD

1. Introduction

NOSQL key-value stores such as BigTable, HBase, and RocksDB have increased in popularity over the past few years, and are now very widely deployed. Yet in many installations, the single-key access offered by these data stores is insufficient; developers increasingly require *ACID* (atomic, consistent, isolated, and durable) *transactions* that access multiple keys. (Transaction API and semantics are discussed in Section 2). This need is addressed by a number of popular *transaction processing systems (TPSs)* such as Percolator [**?**], Omid [**? ?**], Tephra [**?**], and CockroachDB [**?**], which offer transactions on top of BigTable [**?**], HBase [**?**], HBase, and RocksDB [**?**], resp. (In the context of this paper, we consider only TPSs that are layered atop a stand-alone key-value store). The underlying NOSQL stores are typically distributed over multiple *regions* (sometimes called nodes, shards, domains, or partitions). The main purpose of a TPS is to support *global transactions*, i.e., ones that span multiple regions.

The need to atomically perform multiple updates entails a two-phase design, whereby transactions first indicate their *intent* to write certain values, and then atomically *commit* (or *abort*) all their write intents at the same time. In addition, all the values written by a single transaction are associated with a common *timestamp* (or *version number*). Reads, in turn, rely on these timestamps in order to read a consistent snapshot of the data store. They further require some mechanism (either centralized or distributed) to determine what to do on encountering a write intent.

A TPS also implements a *validation* or *conflict detection* mechanism in order to ensure transaction isolation, which can mean either serializability or snapshot isolation (SI) [**?**]; we focus here on the latter (see Section 2) , which is popular in real-world systems and amenable to scalable implementations. Conflicting transactions that would violate the required isolation level are aborted. Conflicts are checked at commit time, and possibly during write-write conflicts.

Algorithm 1 TPS operation schema.

```
 1: procedure BEGIN
 2:     obtain tentative (start) timestamp $ts_1$

 3: procedure READ(key)                          ▷ transactional read
 4:     if hasWriteIntent(key) then
 5:         resolve intents, possibly aborting transactions
 6:     return latest version of key that does not exceed $ts_1$

 7: procedure WRITE(key, value)                  ▷ transactional write
 8:     optionally check for conflicts and abort if found
 9:     indicate write intent for key with value and $ts_1$
10:     add key to local write-set

11: procedure COMMIT(write-set)
12:     obtain commit timestamp $ts_2$
13:     if validate(write-set, $ts_2$) then      ▷ check for conflicts
                                   ▷ commit all write intents with version $ts_2$
14:         atomically and persistently indicate commit
15:     else
16:         abort
17:     clean-up write intents and other meta-data
```

Transaction processing thus follows the general schema outlined in Algorithm 1, while systems vary in their implementations of each of the steps. For example, whereas most systems rely on a centralized service (requiring remote access) for timestamp allocation [**? ? ? ?**], this is not essential [**?**]; similarly, validation (conflict detection) can use a centralized service [**? ? ?**], per-transaction entries in a global table [**?**], or two-phase commit [**?**]. Different ways to implement this schema are discussed in Section 3.

The two-phase transaction processing schema induces high overhead, especially on short transactions. For example, a transaction writing a single key (and reading none) goes through the following stages: (1) *begin* to obtain a tentative timestamp, in most implementations from a centralized service; (2) add *write intent* to the data store; (3) *get commit timestamp*, typically using a centralized service, (though in some cases, the tentative timestamp is used); (4) *validate* (conflict detection), using a centralized service or two-phase locking; (5) *indicate commit*, sometimes at the data store and sometimes at a global table; and (6) *clean-up* – replace write intent with actual write in the data store. The overhead is exacerbated by the fact that some of these steps involve access to centralized services, which, in a large deployment, are remote most of the time.

In this work we expedite *local transactions*, namely, transactions that span a single region, in particular, short ones. We introduce a *fast path* for such transactions, which mitigates this overhead without a significant impact on global (multi-region) transactions. The concept is generic and is applicable to a family of TPSs that use the schema of Section 3, including Percolator [**?**], Omid 1 [**?**], Omid 2 [**?**], Tephra [**?**], and CockroachDB [**?**]. In Section 4 we define API extensions for fast local transactions, and in Section 5, we consider a generic TPS that supports global transactions, and enhance it with optimized support for specific types of local transactions. Our main goal is to expedite short single-object transactions, which are popular in production web workloads [**?**].

A second contribution of this paper is building a new low-latency transaction processing system, Lorra, presented in Section 6, where we embody the fast path concept. Lorra is based on the open-source version of Omid 2 [**?**], a transaction processing engine for HBase, but replaces its centralized commit table access with a lower-latency distributed approach to indicating commits. Lorra supports local transactions via (1) distributed timestamp allocation; (2) extensions to the underlying key-value store API; and (3) an extended client-side transaction library. We implement (1) and (2) in HBase region servers, and (3) based on the Omid client library. [[Idit: Discuss evaluation results given in Section **??**.]]

[[Idit: The roadmap below is not essential; maybe replace with summary of contributions.]] The remainder of this paper is organized as follows: We begin by providing the context for this work: in Section 2 we define the API and semantics of a TPS, and Section 3 surveys the design space of popular TPS implementations, highlighting the common properties of TPSs that can benefit from our approach. In Section 4 we propose our extended TPS API, which provides a fast-path for local transactions. Section 5 then describes our algorithm for supporting fast-path local transactions. Section 6 presents Lorra, our low latency transaction processing system, where our fast-pass service is implemented, and Section **??** presents its evaluation. We review related work in Section **??** and conclude with Section **??**.

2. Transaction API and Semantics

We consider a TPS that spans multiple regions (sometimes called nodes, domains, or shards) and supports global (multi-region) transactions. Data is stored in an underlying NOSQL key-value store, such as HBase [**?**], BigTable [**?**], or RocksDB [**?**]. Clients access the data store directly using the TPS API, and partake in transaction coordination, possibly using the assistance of a centralized service.

We first describe the data model and API of the underlying data store (Section 2.1). We then proceed to define the transaction semantics provided by the TPS (Section 2.2).

2.1 Data store

The underlying data store holds *objects* (often referred to as *rows*) identified by unique *keys*. 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. Thus, at any given time, an object holds a tuple ⟨key,⟨version,value⟩+⟩, where value can be structured to consist of multiple columns. We further assume that a write operation can specify the version number it writes to. The underlying data store provides the following API:

⟨version,value⟩ read(key) – atomically returns the value with the highest version associated with key along with its version.

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

write(key,value,version) – atomically creates or updates the version: if the version already exists, its value is updated; otherwise, a new version is added. Garbage collection of obsolete versions is a separate process.

Read-modify-write – data stores often provide means to atomically read and update an object, (e.g., HBase exports CheckAndMutate operations, which are internally implemented using a RW lock, whereas BigTable supports row transactions). We will extend this capability below in order to implement certain atomic operations at the data store level.

2.2 Transaction semantics

A *transaction* is a sequence of read and write operations on different objects that ensures the so-called ACID properties: *atomicity* (all-or-nothing execution), *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 systems providing snapshot isolation [**?**], which is provided by popular database technologies such as Oracle, PostgreSQL, and SQL Server.

Intuitively, SI ensures that the information a transaction retrieves from the database does not mix old and new values. For example, if a task 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. More precisely, 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-versioned concurrency control) that allow more concurrency than serializable ones, and hence scale better.

TPSs allow programmers to delineate transactions via the begin and commit APIs: the sequence of read and write operations a client invokes between begin and commit pertains to one transaction. 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. An abort may also be initiated by the programmer, e.g., on encountering an error. Applications typically retry a transaction upon (either type of) abort.

The data is *partitioned* (or sharded), and each object belongs to one region. Global transactions may span multiple regions, and atomically commit or abort on all.

3. Context and Applicability

This section outlines the general characteristics of transaction processing systems that can benefit from our suggested fast-path for handling short local transactions; these TPSs follow the generic schema of Algorithm 1 above. We illustrate how Percolator [**?**], Omid 1 [**?**], Omid 2 [**?**], Tephra [**?**], and CockroachDB [**?**] adhere to this schema.

A TPS's API is offered by a client library, which accesses the data directly in the data store, and performs coordination actions to begin, commit, or abort transactions.

The TPSs we consider all rely on timestamps for coordination: when a transaction begins it obtains a *tentative timestamp* ts_1, which it uses (1) to determine what version of each key to read; possibly (2) as a tentative version for data items it writes; and possibly (3) as a unique transaction id. In Percolator, Omid, Tephra, and Lorra, timestamps are provided by a monotonically increasing *global version clock (GVC)* managed by a central service. CockroachDB instead uses the local clock of one of the regions (called node therein), which is closely synchronized relative to other regions and furthermore ensures causality. We use a similar approach for local transactions in Lorra.

Upon commit, a transaction obtains a *commit timestamp* ts_2, and the items it wrote are persisted with this version. Snapshot isolation allows the commit timestamp to be later than the tentative timestamp used for reading, which TPSs exploit to reduce conflicts.

During a transaction, write calls indicate their intents to write; this is done using a dedicated column (part of the value in our data model). A write intent can take the form of a semantic lock on the key, blocking read attempts (as in Percolator) or an indication that the write is uncommitted and hence can be safely ignored (as in Omid). While write intents differ across implementations, we assume that the following generic boolean function is provided:

hasWriteIntent(key, version) returns true if the specified version of key has a write intent indication; if version is not provided, the latest version associated with key is checked.

In more detail, a transaction goes through the following phases (outlined in Algorithm 1):

1. *Begin* – When a transaction begins, it obtains a tentative timestamp (version) ts_1 for reading its consistent snapshot, and unique transaction id. The two can be combined (i.e., ts_1 can serve as the transaction id, provided that it is unique). In most cases, this is done using a centralized entity (sometimes called timestamp oracle [**? ?**] or transaction manager [**?**]). In CockroachDB, the timestamp is based on a local clock that is "close to" real-time and preserves causality across regions, and unique transaction ids are used to break ties in case timestamps (from different regions) are identical.
2. *Transactional reads and writes* – In the course of a transaction, reads obtain a consistent snapshot, while write operations indicate their *intent* to write.
   - *A read operation* obtains the value of a single object atomically. Only versions committed with timestamps smaller or equal to the transaction's ts_1 are read. Different policies are used for resolving pending write intents. For example, Omid 1 resolves them using data on all past commits that the client obtains at begin time, whereas CockroachDB and Omid 2 have the read refer to a centralized transaction table (also called commit table). Percolator waits for the intent to be lifted (by the completion of the writing transactions), and in case the wait times-out, forces this transaction to abort. CockroachDB forces the transaction with the write intent to either commit with a higher version than its own read timestamp, or abort. Similarly, the solution we implement in Lorra forces the transaction with the pending write intent to abort.

- *Indicating write intents* changes the objects' write intent columns to indicate a transaction attempting to write them is under way. The value the transaction intends to write is added with a new version, typically the transaction's ts_1. Each object is updated atomically by itself. In some solutions writes check for conflicts before declaring their intents [?], whereas in others, all conflict detection is deferred to commit time.

3. *Obtain commit timestamp* – When a transaction is ready to commit, it gets a commit timestamp, ts_2. In most cases, e.g., [? ? ? ?], this is the GVC value at commit time.
4. *Validation/conflict detection* – Before a transaction can commit, it must check that it does not conflict with any concurrent transactions. For SI, we need to check for write-write conflicts only. This step checks write intentions as well as version numbers. It may be combined with the above step using a centralized transaction manager [? ? ?], or may rely a two-phase commit protocol where the written objects are first locked, then validated, and then committed [?]. Alternatively, conflicts may be detected in situ by atomically validating each written object as part of adding a write intent [?].
5. *Indicate commit* – In case validation succeeds, the transaction is committed in one atomic irrevocable step. This is achieved by writing to a designated persistent *commit entry*, which can reside in a global table (like the commit table of Omid 2 and the transaction table of CockroachDB) or alongside the first key written by the transaction (as in Percolator and Lorra). Note that a commit attempt may fail and abort instead.
6. *Clean-up* – Finally, a transaction changes its write intents to persistent writes in case of commit, and removes them in case of abort. This phase occurs after the transaction is persistently committed or aborted via the commit entry, in order to reduce the overhead of future transactions (by sparing them the need to check the commit entry) and to garbage collect obsolete information.

4. Introducing Fast Path Local Transactions

The main goal of our fast path is to forgo the overhead associated with two-phase transaction processing, especially for short transactions. In particular, the fast path can eliminate the need for remote access in local (single-region) transactions and can reduce the number of calls (RPCs) the client performs. To this end, we introduce in Section 4.1 a streamlined API that jointly executes multiple API calls of the original TPS. We refer to transactions that use this API as *fast path (FP) transactions*. We discuss the semantics of FP transactions relative to regular ones in Section 4.2.

4.1 API and usage

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

br(key) – begins an FP transaction and reads key within it. Recall that read-only transactions don't need to commit.

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

A more elaborate example is a read-modify-write API:

brwc(key,f) – begins an FP transaction, reads the latest version of key, applies f to it (on the server side), writes the result into a new version of key that exceeds all existing ones, and commits.

To use the above API, the programmer has to encapsulate the transaction logic in a function for server-side processing. Alternatively, we allow FP transactions to unfold dynamically much like regular transactions do. A dynamic FP transaction may instead begin with a br call, perform client-side processing, and then call the following function to update either the same or a different key:

wc(key,val) – writes val into a new version of key that exceeds all existing ones, and commits.

Moreover, we do not restrict FP transactions to perform a single read – any number of r's may be called between the br and wc. The supported types of FP transactions are summarized in Table 1. Note, however, that all calls must be directed at the same region, else the transaction is not local. In case an FP transaction dynamically discovers that it needs to access additional regions, it is aborted and should be restarted as a regular transaction.

Call sequence	Transaction type
br	single read
bwc	single write
br, r*	multi-read
br, r*, wc	multi-read, single write
brwc	server-side single-read-write

Table 1: Supported FP transaction types.

In principle, it would have been possible to also allow w calls in the span of an FP transaction, but in this case, it is not possible to forgo the two-phase execution. That is, the w calls would need to indicate write intents, and to be atomically committed (or aborted) during the final wc (or c) call. Given the limited benefit and extra complexity of allowing many writes in FP transactions, we do not support this option in our solution.

4.2 Semantics

The semantics for ordering FP transactions relative to regular ones are weaker than SI in that they do not guarantee real-time order over all regular and FP transactions together. Specifically, a regular transaction overlapping two FP ones that access different regions may observe an update of the second and miss an update by the first. For example, assume objects x and y are managed in two different regions, then real-time order can be violated as illustrated in Figure 1 (ignore the skip operations for now; they will be explained in the next section).

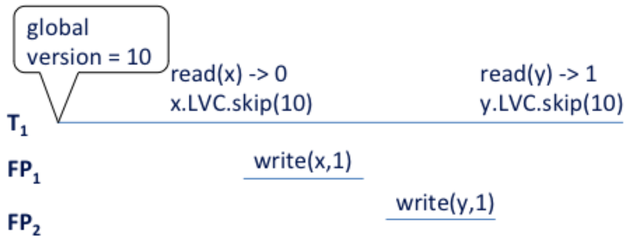


Figure 1: Possible violation of real-time order among fast path (FP) transactions in different regions. Global transaction T_1 reads x before it is updated by FP transaction FP_1 and reads y after it is updated by FP transaction FP_2 even though FP_2 occurs after FP_1. $T1$'s global version is 10, and its skips the local version clocks of the regions holding x and y to 10 when reading from them.

The system still 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. FP transactions within each region are ordered in $\mathcal{T}$ according to their commit times;
3. each transaction's read operations see a consistent snapshot of the database reflecting a prefix of $\mathcal{T}$ that includes all transactions committed prior to its start time plus any number of concurrent FP transactions; and
4. a transaction commits only if none of the items it updates has been modified since that snapshot.

Note that since $\mathcal{T}$ respects commit times in each region, causality is preserved, because only transactions that access disjoint sets of data objects can be re-ordered.

5. Fast Path Algorithm

We now explain our support for local transactions with SI semantics in the context of a system as described in Section 3 above. Our solution consists of three parts: First, in Section 5.1, we enhance the underlying data store with support for per-region *Local Version Clocks (LVCs)*. This aspect is already implemented in CockroachDB, which uses per-region Hybrid Logical Clocks [?] in order to allow for distributed timestamp allocation. In the systems that maintain a GVC, (e.g., [? ? ? ?]), our addition of LVCs entails a minor modification to management of global timestamps (in the transaction manager or oracle). Second, in Section 5.2 we extend the underlying data store's API to allow manipulating a region's LVC jointly with objects stored at that region. Finally, we add client-side support for the fast path API, as explained in Section 5.3.

5.1 Local version clock

Like the GVC, LVCs are also monotonically increasing. Each region has its own LVC, which is loosely synchronized with the GVC. The idea is to use the region's LVC for ordering FP transactions in any given region, and allow FP transactions to progress independently in different regions. Note that it is safe to do so because no global order needs to be enforced among FP transactions that access disjoint sets of objects.

Regular (multi-region) transactions, in turn, continue to obtain their versions from the GVC (except in CockroachDB, where local clocks are used for all transactions and a conservative conflict detection policy enforces aborts in case there is uncertainty regarding the global order of events). Therefore, whenever a regular transaction T accesses a given region, we synchronize that region's LVC with the GVC in order to ensure that the timestamp obtained by T exceeds those obtained by earlier completed transactions within the region, and that later transactions within that region will obtain higher versions than T's. (A similar approach was used for enforcing causality in Hybrid Logical Clocks [?] as used by CockroachDB, and for supporting local transactions in Mediator [?].

To support such loose synchronization, the GVC now advances at a coarse granularity of *epochs*. This can be implemented, for example, by choosing some epoch size 2^ℓ, and keeping the ℓ least significant bits of the GVC padded with zeros. In other words, every increment of the GVC increases its value by 2^ℓ. The LVC obtains the epoch ($n - \ell$ most-significant bits for an n-bit GVC) from the GVC, and proceeds to assign timestamps within the designated epoch (by incrementing the least significant bits).

The LVC has a single component, LVC.current, and it supports the following API:

LVC.skip(epoch) – atomically set LVC.current to max(epoch, LVC.current).

LVC.fetchAndIncrement() – atomically increment LVC.current and return its new value. For simplicity, we assume that the LVC is used so it does not overrun the epoch and does not wrap around within the epoch. That is, fetchAndIncrement is called less than 2^ℓ times in each epoch.

LVC.get() – return LVC.current.

By incrementing the GVC, a multi-region transaction essentially initiates a new epoch, and obtains a timestamp exceeding all those of older local transactions. In addition, multi-region transactions enforce the synchronization of the LVC with respect to the GVC using the skip operation. Specifically, whenever a transaction in a new epoch accesses (for either read or write intention indication) an object in a region whose LVC is still in an older epoch, it invokes that region's LVC.skip so it will not lag behind the transaction's tentative timestamp (ts_1) obtained from the GVC. Thus, new local transactions that will begin later in the region will have higher timestamps, as needed. Note that transaction commits do not alter the LVC; the LVC only reflects transactions' tentative timestamps obtained when they begin.

Note also that as long as a running transaction does not access a given region, further updates can occur by local transactions in that region, and these updates can be reflected in the transaction's snapshot in case it later reads these objects. Thus, unlike with regular transactions, which satisfy real-time order, a transaction's snapshot may reflect changes that occur after it commences, as in the example of Figure 1.

Here, x is written with LVC=10 in one region, and later y is written in another region with its LVC=0. The concurrent transaction's snapshot time is 10, which includes the update of x and not that of y. The same scenario does not occur with two local transactions accessing the same region, since once the LVC is incremented, no further updates in the same region can occur with older LVC values.

5.2 Accessing data and the LVC together

In order to allow FP transactions to execute with a single data store access, we extend the data store to support functions that access data objects and the LVC together. Thus, an FP transaction can increment LVC, obtain a timestamp, write with this timestamp, and commit, all in one round-trip to the local data store.

We further enforce atomic access to the data and the LVC. This is important in order to avoid races between obtaining a version from the LVC and updating the data. For example, if the updates of the LVC and the data were separate, the following scenario could have arisen (x and y are in the same region): [Yoni: Find shorter example?]

- FP transaction FP_1 plans to update object x and obtains LVC value 1.
- Regular transaction T_1 obtains tentative timestamp 10. [Yoni:
- T_1 reads y, causes LVC to skip to 10.]
- FP transaction FP_2 obtains LVC value 11 and updates object y.
- Regular transaction T_2 obtains tentative timestamp 20.
- T_2 reads the old version of x (since it had not yet been written by FP_1) and the new version of y.
- FP_1 writes x with version 1 and completes.
- T_2 reads the new version of x and the old version of y.

Here, SI is violated.

In addition, the new functions must take care not to breach the atomicity of concurrent transactions. To this end, they rely on write intents. One option is to abort an FP transaction whenever a write intent exists for one of its written or read keys.[Yoni: read keys?] Starvation can be avoided by retrying the transaction as a regular one. Alternatively, an FP transaction may resolve write intents similarly to regular ones. Note that in case the resolution accesses a global table, this makes the FP transaction non-local. Nevertheless, such remote access is needed only in case of contention between a regular transaction and a concurrent FP one on the same key.

We extend the data store API with the following calls, whose pseudocode is given in Algorithm 2.

getWithTs(key) returns the current LVC read ts, plus the highest committed version-value pair pertaining to key with version up to ts. Write intents are either resolved or lead to abort.

mutate(key, ts) atomically increments the region's LVC and creates a new version for key associated with the new LVC value. The operation fails (returns false) if the latest version of the object has a write intent indication.

validateAndMutate(key, ts, value) atomically validates that the highest version of key in the data store does not exceed the provided timestamp and mutates the object associated with key and the LVC as mutate does. Returns true if the validation is successful, otherwise returns false and does not perform the mutation.

5.3 Client-side support for FP transactions

There are two aspects to supporting FP transactions. First, we need to implement the client API defined in Section 4 above. We do this in Section 5.3.1. Second, we need to modify the implementation of regular transactions to be aware of FP ones; this is addressed in Section 5.3.2.

5.3.1 Implementing FP APIs

Pseudocode for the client-side implementation of the FP transaction API is given in Algorithm 3. As for regular transactions, the ongoing transaction's tentative timestamp is stored in a local variable, which we denote by ts_1.

The br and bwc APIs are implemented directly using the new functions getWithTs and checkAndMutate, resp. In addition, br stores its obtained timestamp in ts_1, for potential subsequent r calls. An ensuing wc call validates that the written key has not been modified since the preceding br call.

Another streamlined type of FP transaction is a single-key read-and-write, brwc. It is parametrized by some compute function f that generates the new value from the old one. Because brwc does not perform the update to the data store atomically with the read, it needs to validate the mutate operation to ensure that the object has not been updated since

Algorithm 2 Atomic access of LVC and data in a single region data store. Each method executes atomically.

```
procedure GETWITHTS(key)
    ts ← LVC.get()
    ⟨ version, value ⟩ ← highest version $\leq$ ts of key
    if hasWriteIntent(key, version) then
        resolve write intent or abort
    return ⟨ ts, value ⟩

procedure MUTATE(key, value)
    if hasWriteIntent(key) then
        return false
    version ← LVC.fetchAndIncrement()
    add ⟨ version, value ⟩ to key
    return true

procedure VALIDATEANDMUTATE(key, ts, value)
    if hasWriteIntent(key) then
        return false
    if latest version of key $>$ ts then
        return false
    version ← LVC.fetchAndIncrement()
    add ⟨ version, value ⟩ to key
    return true
```

its latest version was read at the beginning of the operation. If the validation fails, the transaction can be retried, either the same way or using a regular transaction.

Algorithm 3 Client-side code for FP transactions.

```
local variable $ts_1$          ▷ tentative ts of ongoing transaction

procedure BR(key)
    ⟨ $ts_1$, value ⟩ ← getWithTs(key)
    return value

procedure BWC(key, value)
    return mutate(key, value)

procedure WC(key, value)
    return validateAndMutate(key, $ts_1$, value)

procedure BRWC(key, f)
    ⟨ $ts_1$, value ⟩ ← getWithTs(key)
    return validateAndMutate(key, $ts_1$, f(value))
```

5.3.2 Ensuring monotonicity of written versions

We saw that FP transactions become aware of pending regular transactions via write intents. This ensures correctness in case a write intent is written for a key before the FP transaction accesses it. However, it may be the case that a regular transaction with a smaller timestamp than the FP transaction will write its intent (with the smaller timestamp) *after* the FP transaction accesses the key as in the following scenario:

1. Transaction T_1 begins with tentative timestamp 10.
2. FP transaction FP_1 writes key and commits with timestamp 11.
3. Transaction T_1 writes key with timestamp 10.

In this case, T_1 must detect the write-write conflict with FP_1 and abort. To this end, we slightly modify the write operation of regular transactions so as to validate that no later versions exist for the written key. In other words, a write operation only creates a new version if it exceeds all existing ones, and otherwise aborts. We note that some data stores and TPSs already employ this modus operandi of writing only monotonically increasing versions. This feature is also beneficial for regular transactions, as it leads to early conflict detection.

6. Low-latency Transactions with Lorra

To realize our local transactions, we build Lorra, a scalable low-latency TPS on top of HBase. Lorra is an evolution of Omid 2, redesigned to reduce latency by distributing the commit entries. We describe Lorra's normal transaction processing in Section 6.1, and then proceed to discuss our implementation of fast path transactions in Section 6.2.

6.1 Lorra design choices

In Section 3 we outlined the design space of existing TPSs that support SI semantics via the general paradigm outlined in Algorithm 1. The design choices made by different existing TPSs as well as by Lorra are summarized in Table 2. [Yoni: Does centrelized mean only one entity has access to it, or does it mean the table is not distributed?]

TPS	txn	validation		reads
	entry	scheme	time	cause abort
Omid, Tephra	**R**	**C**	commit	no
Percolator	**D**	**D**	commit	yes
Omid 2	**C**	**C**	commit	no
CockroachDB	**C**	**D**	write	yes
Lorra	**D**	**C**	commit	[Yoni: yes?!]

Table 2: Design choices in TPSs. **C** – centralized, **D** – distributed, **R** – replicated.

All considered TPSs store per-transaction 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, and are checked in order to resolve write intents in line 5. Omid 1 and Tephra replicate this information among all active clients, which consumes high bandwidth and does not scale. Omid 2 and CockroachDB use dedicated tables. In experiments we ran, we observed that the centralized table is Omid 2's principal scalability bottleneck, and while this bottleneck is mitigated via batching commit table updates, this also increases latency. We therefore opt to follow (in this aspect) the design of Percolator, which stores the transaction's entry alongside its first written key and stores pointers to this first key at all other keys written by the transaction. Thus, updates of transaction entries can be performed in parallel by independent clients, and such updates are no longer a performance bottleneck. We will see below that this modification improves performance [[Idit: quantify!]]

Percolator also performs conflict detection (line 13) in a distributed manner, using a two-phase commit protocol, where all objects are locked during the validation. Unfortunately, as long as a lock is held, it blocks all transactions attempting to access the locked object. Since clients might stall or crash while holding a lock, Percolator supports a recovery procedure that allows blocked transactions to forcefully abort pending ones, after a certain timeout. Later works (on Omid and Tephra) have decided to forgo such blocking, and replaced the two-phase commit protocol by centralized conflict detection. CockroachDB takes a different approach of performing 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 the centralized conflict detection in Omid 2 is extremely scalable, (capable of sustaining orders of magnitude higher throughput than the network), Lorra adopts this mechanism from Omid 2.

The centralized service, called Transactional Status Oracle (TSO), maintains an in-memory hash table mapping keys to commit timestamps (ts_c) of transactions that last wrote them. A conflict arises whenever a key in a transaction's write-set has been written with a timestamp higher than its ts_r. In case the conflict detection service crashes, all pending transactions are aborted, and it can be immediately restarted with an empty table, because only conflicts with concurrent transactions need to be checked.

Whenever a read encounters a write intent, it resolves it (line 5) via the commit entry at the first written key. If the commit entry is committed, the written value is taken into account, and if it is aborted, the value is ignored. But if the commit entry is neither committed nor aborted, the reading transaction forces the writing transaction to abort by using an atomic read-modify-write operation to set its status to aborted. Similar scenarios occur in Percolator and CockroachDB, since they use distributed validation. Omid and Tephra, on the other hand, do not need to force such aborts, because they use a single centralized service for timestamp allocation, validation, and writing the commit entry in a way that ensures that if the read sees a write intent by an uncommitted transaction, that transaction will not commit with an earlier timestamp than the read. The possibility of reads aborting pending transactions means that commit attempts (line 14) have to check whether the transaction is aborted atomically with writing the commit status.

key	value	version	commit	leader
k1	a	3		
k2	b	3		k1
k2	c	3		k1

(a) Tentative transaction

key	value	version	commit	leader
k1	a	3	7	
k2	b	3		k1
k2	c	3		k1

(b) Committed transaction

key	value	version	commit	leader
k1	a	3	7	
k2	b	3	7	
k2	c	3	7	

(c) Post-committed transaction

Figure 2: Different stages of metadata during a transaction. The $txid$ the transaction received from the begin stage is 3, and the ts_c it received when committing is 7. The leader chosen for this transaction is k1

Algorithm 4 Lorra's *get(key)* operation.

```
procedure GET(KEY)
    for rec ← ds.get(key, versions down from $ts_r$) do
        if rec.commit≠nil then                          ▷ not tentative
            if rec.commit < $ts_r$ then
                return rec.value
        else                                            ▷ tentative
            value ← GETTENTATIVEVALUE(REC)
            if value ≠nil then
                return value

procedure GETTENTATIVEVALUE(REC)
    leader ← rec.leader
    $tx_c$ ← checkAndInvalidate(leader)
    if $tx_c \neq$ nil then                              ▷ committed
        update rec.commit                               ▷ helping
        if $tx_c < ts_r$ then return rec.value
    else
        return nil
```

6.2 Local transactions in Lorra

To support local transactions, we implemented the algorithms and APIs described in Section 5. Each Hbase region server maintains an LVC which is updated whenever a transaction accesses it. Omid 2 API was extended to support FP transactions, and the write stage in reglar transactions was modified to use checkAndMutate instead of put.

7. Lorra Implementation

[Yoni: explain here or in ?? that each key has multiple versions, we use txd tx_r tx_c....]

7.1 Transaction metadata

[Yoni: make sure Idit defines in section **??**] Figure 2 shows the metadata of a transaction during its stages of execution. For each version of a key in the data store we add 3 additional fields: (1) A *version* field, which holds the $txid$ of the transaction that wrote it, in our implementation the tx_r received at the begin stage is used as the $txid$. (2) A *commit* field, which indicates whether the data is tentative or committed, if it's committed, the value will be the transaction's ts_c. (3) A *leader* field. Since the transaction commit needs to be an atomic step, one key from the transaction's write set is chosen to be the *leader* of that transaction, and when the transaction is committed a single atomic write is initiated to the *leader's* commit field with the transaction's ts_c. Transaction read operations refer to the key's *leader* to find out whether the value has been committed or not.

7.2 Client-side operation

Begin The client obtains from the TSO a read timestamp ts_r, which also becomes its transaction id ($txid$).

Get Algorithm 4 describes Lorra's implementation of Algorithm 1 READ procedure. The data store's get operation is referred to as ds.get(key,version). The algorithm traverses records pertaining to key with versions smaller than ts_r, latest to earliest, and returns the first key that is committed. Upon encountering a tentative record (with commit=nil), the algorithm will get the leader's record from the data store, read its ts_c from the *commit* field and return the key if the ts_c is smaller than ts_r. To ensure SI, if the leader's *commit* field is nil, the transaction that leader represents must be aborted [Yoni: example here?]. The checkAndInvalidate function will atomically read the *commit* field and if its value is nil, will insert an invalidation symbol in it. When a transaction commits, it atomically checks it hasn't been invalidated before inserting the ts_c to the *commit* field.

Put The client adds the tentative record *(key, value, txid, nil,leader)* to the data store. The leader is chosen to be the first key the client updates during a transaction. To ensure monotonicity of written versions as seen in Section 5.3.2, instead of using the standard *put* operation the data store exposes, we must use checkAndMutate which will atomically put the record only if the latest version of the key is smaller than ts_r.

Commit The client requests *commit(txid, write-set)* from the TSO. The TSO assigns it a commit timestamp ts_c and checks for conflicts. If there are none, the client will write the ts_c in the leader's *commit* field. As explained at the Get stage, the transaction must atomically commit only if no other get invalidated it. This is done using the data store checkAndMutate function to read the invalidation symbol before writing the ts_c. To avoid an extra RPC to the leader on every read, after a transaction is committed, a post-commit stage writes the ts_c to the *commit* field of every key in the write set of that transaction. Following a successful commit, the client adds ts_c to all data items it wrote to.

Singleton Read [Yoni: This is the same as 5.3.1]

Singleton Write [Yoni: what about WC and BRWC?]

7.3 HBase region server

[Yoni: is there something special to say here?]

8. Evaluation

This section reports the evaluation results of Lorra.

Methodology To evaluate Lorra, we set up a 3 node Hbase cluster loaded with 50GB of data, and a machine hosting the TSO. Another machine created load on the system by running threads that generated random transactions towards Hbase and the TSO. To create further load on th TSO, we

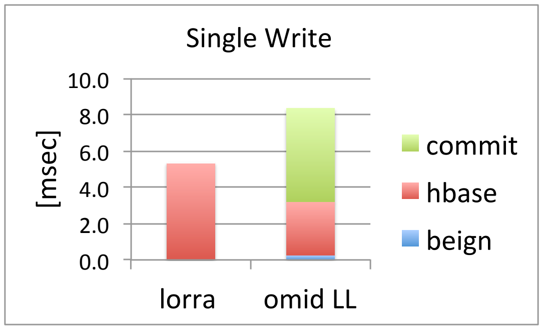
Transaction size had a zipfian distribution from 1 to 10. To measure the latency of transactions we modified YCSB [**?**] to perform a begin and commit towards the TSO at the beginning and end of each transaction, and used YCSB framework to measure the latency of each stage.

Distributed txn entry We first compare Lorra's distributed txn entry approach to Omid 2 centralized approach. Figure **??** shows the latency of a transaction while applying increasing load on the system. The load is measured in kilo transactions per second (KTPS), and the transaction latency is measured in milliseconds. Figure **??** shows the latency of a transaction with a single read or write. The figure shows that latency obtained by Lorra is $4\times$ faster than Omid 2. Lorra performs better because Omid 2 is throughput oriented and batches the writes to txn entries, so on every begin stage the client has to wait for all commits in the batch to get persistent. Figure **??** shows the latency of transactions with 5 read or write operations. This time the latency obtained by Lorra is $1.8\times$ faster than Omid 2. For transactions with 5 operations to an Hbase table, the begin and commit time become negligible.

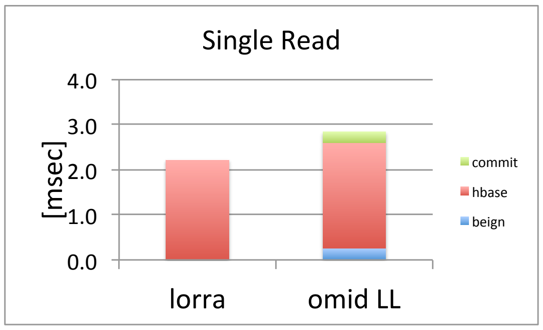
FP API latency Figure **??** compares the latency observed by YCSB when running regular and FP transactions. We focus on 3 types of transactions: (1) Read, (2) Write and (3) Read modify write.

Figures **??** and **??** show the latency of a single read and write transaction. Regular transactions (b-r-c, b-w-c) query the TSO at the begin and commit stage which add 0.3ms for read transactions and 3ms for write transactions. Figure **??** compares the latency of a read modify write transaction, once using the FP API (br-bwc) and once using the regular transaction API (b-r-w-c).The begin and commit stage of the regular transaction add 3ms to each transaction.

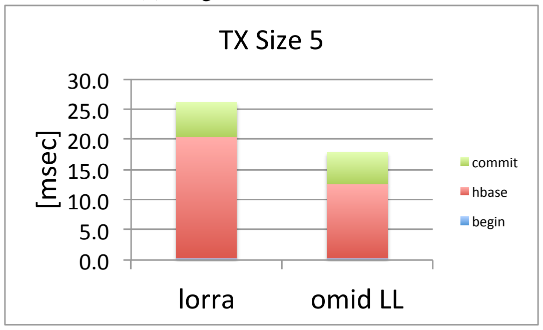
[Yoni: –background noise –switch order of charts –explain that the LL is better for all TPS so it doesnt matter the point we choose to check]



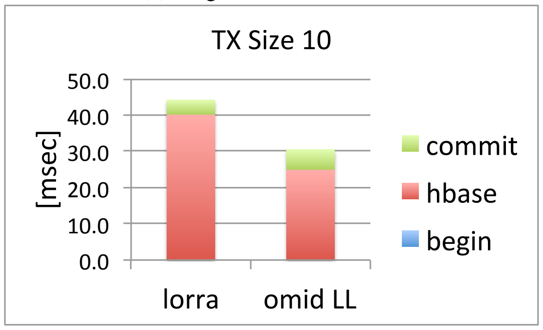
(a) Single write transactions



(b) Single read transactions



(c) Transaction size 5



(d) Transaction size 10

Figure 3: Latency of Lorra and omidLL

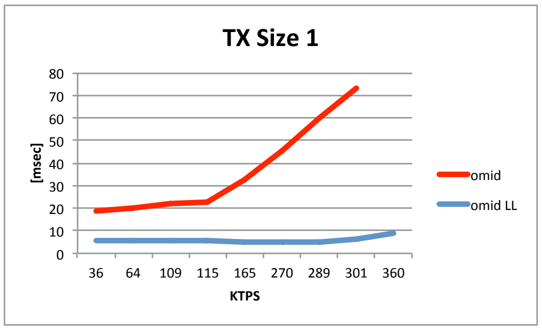
[[Idit: Compare the following: Omid 2, regular transactions in Lorra, local transactions in Lorra, HBase.]]

[[Idit: Add graph showing cost of RMW (validation) in regular transaction write.]]

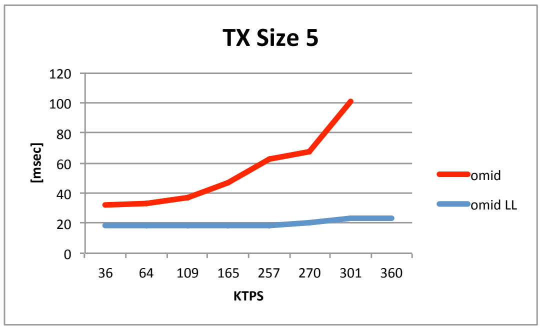
9. Related Work

10. Conclusion

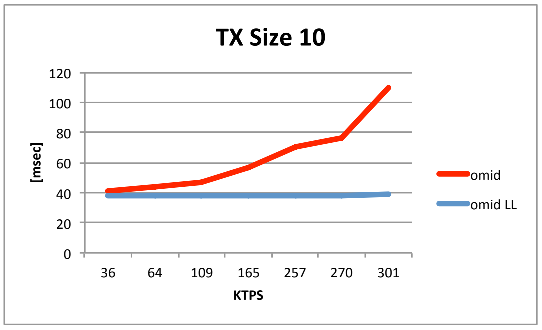
A. Notations and definitions



(a) Transaction of size 1 latency



(b) Transaction of size 5 latency



(c) Transaction of size 10 latency

Figure 4: A Comparison between Omid 2 centralized txn entry and Lorra's distributed txn entry.