Transactional storage for geo-replicated systems

Yair Sovran* Russell Power* Marcos K. Aguilera† Jinyang Li*
*New York University †Microsoft Research Silicon Valley

ABSTRACT

We describe the design and implementation of Walter, a key-value store that supports transactions and replicates data across distant sites. A key feature behind Walter is a new property called *Parallel Snapshot Isolation* (PSI). PSI allows Walter to replicate data asynchronously, while providing strong guarantees within each site. PSI precludes write-write conflicts, so that developers need not worry about conflict-resolution logic. To prevent write-write conflicts and implement PSI, Walter uses two new and simple techniques: preferred sites and counting sets. We use Walter to build a social networking application and port a Twitter-like application.

Categories and Subject Descriptors: C.2.4 [Computer-Communication Networks]: Distributed Systems—client/server; distributed applications; distributed databases; D.4.5 [Operating Systems]: Reliability—fault-tolerance; H.3.4 [Information Storage and Retrieval]: Systems and Software—distributed systems

General Terms: Algorithms, Design, Experimentation, Performance, Reliability

Keywords: Transactions, asynchronous replication, geo-distributed systems, distributed storage, key-value store, parallel snapshot isolation

1. INTRODUCTION

Popular web applications such as Facebook and Twitter are increasingly deployed over many data centers or *sites* around the world, to provide better geographic locality, availability, and disaster tolerance. These applications require a storage system that is *geo-replicated*—that is, replicated across many sites—to keep user data, such as status updates, photos, and messages in a social networking application. An attractive storage choice for this setting is a key-value store [16], which provides good performance and reliability at low cost.

We describe Walter, a geo-replicated key-value store that supports *transactions*. Existing geo-distributed key-value stores provide no transactions or only restricted transactions (see Section 9). Without transactions, an application must carefully coordinate access to data to avoid race conditions, partial writes, overwrites, and other hard problems that cause erratic behavior. Developers must address these same problems for many applications. With transactions, developers are relieved from concerns of atomicity, consistency, isolation, durability, and coordination. For example, in a social networking application, one may want to remove user A from B's friends list and vice versa. Without transactions, developers must write code carefully to prevent one removal from happening without the other. With transactions, developers simply bundle those updates in a transaction.

Transactions in Walter ensure a new isolation property called *Parallel Snapshot Isolation* (PSI), which provides a balance between consistency and latency [22, 54], as appropriate for web applications. In such applications, a user might log into the site closest to her, where she accesses application servers, ad servers, authentication servers, etc. These hosts should observe a consistent storage state. For example, in a social network, a user expects to see her own posts immediately and in order. For that reason, the storage system should provide a strong level of consistency among hosts in her site. Across sites, weaker consistency is *acceptable*, because users can tolerate a small delay for their actions to be seen by other users. A weaker consistency is also *desirable*, so that transactions can be replicated across sites asynchronously (lazy replication).

Eventual consistency [44, 47] is often the property provided by asynchronous replication. When different sites update the same data concurrently, there is a conflict that must be resolved by application logic. This logic can be complex, and we want to avoid forcing it upon developers.

With PSI, hosts within a site observe transactions according to a consistent snapshot and a common ordering of transactions. Across sites, PSI enforces only causal ordering, not a global ordering of transactions, allowing the system to replicate transactions asynchronously across sites. With causal ordering, if Alice posts a message that is seen by Bob, and Bob posts a response, no user can see Bob's response without also seeing Alice's original post. Besides providing causal ordering, PSI precludes write-write conflicts (two transactions concurrently writing to the same object) so that developers need not write conflict resolution logic.

To prevent write-write conflicts and implement PSI, Walter relies on two techniques: *preferred sites* and *counting sets*. In web applications, writes to an object are often made by the user who owns the object, at the site where this user logs into. Therefore, we assign each object to a *preferred site*, where objects can be written more efficiently. For example, the preferred site for the wall posts of a user is the site closest to the user. Preferred sites are less restrictive than primary sites, as we discuss in Section 2.

Preferred sites may not always suffice. For example, a friends list can be updated by users in many sites. The second technique in Walter to avoid conflicts is to use a new simple data type called a counting set (cset), inspired by commutative data types [29]. A cset is like a set, except that each element has an integer count. Unlike sets, csets operations are commutative, and so they never conflict [25]. Therefore, transactions with csets can commit without having to check for conflicts across sites. When developing applications for Walter, we used csets extensively to store friend lists, message walls, photo albums, and message timelines. We found that csets were versatile and easy to use.

Walter uses multi-version concurrency control within each site, and it can quickly commit transactions that write objects at their preferred sites or that use csets. For other transactions, Walter resorts to two-phase commit to check for conflicts. We found that the

latter type of transaction can be avoided in the applications we built.

Using Walter as the storage system, we build WaltSocial, a Facebook-like social networking application, and we port a third-party Twitter-clone called ReTwis [2]. We find that the transactions provided by Walter are effective and efficient. Experiments on four geographic locations on Amazon EC2 show that transactions have low latency and high throughput. For example, the operation to post a message on a wall in WaltSocial has a throughput of 16500 ops/s and the 99.9-percentile latency is less than 50 ms.

In summary, our contributions are the following:

- We define Parallel Snapshot Isolation, an isolation property well-suited for geo-replicated web applications. PSI provides a strong guarantee within a site; across sites, PSI provides causal ordering and precludes write-write conflicts.
- We describe the design and implementation of Walter, a geo-replicated transactional key-value store that provides PSI. Walter can avoid common write-write conflicts without cross-site communication using two simple techniques: preferred sites and csets.
- We give distributed protocols to execute and commit transactions in Walter.
- We use Walter to build two applications and demonstrate the usefulness of its transactional guarantees. Our experience indicates that Walter transactions simplify application development and provide good performance.

2. OVERVIEW

Setting. A geo-replicated storage system replicates objects across multiple sites. The system is managed by a single administrative entity. Machines can fail by crashing; addressing Byzantine failures is future work. Network partitions between sites are rare: sites are connected by highly-available links (e.g., private leased lines or MPLS VPNs) and there are redundant links to ensure connectivity during planned periods of link maintenance (e.g., using a ring topology across sites). We wish to provide a useful back-end storage system for web applications, such as social networks, web email, social games, and online stores. The storage system should provide reliability, a simple interface and semantics, and low latency.

Why transactions? We illustrate the benefit of transactions in a social networking application, where users post photos and status updates, befriend other users, and write on friends' walls. Each site has one or more application servers that access shared user data. When Alice adds a new photo album, the application creates an object for the new album, posts a news update on Alice's wall, and updates her album set. With transactions, the application groups these writes into an atomic unit so that failures do not leave behind partial writes (atomicity) and concurrent access by other servers are not intermingled (isolation). Without transactions, the application risks exposing undesirable inconsistent state to end users. For example, Bob may see the wall post that Alice has a new album but not find the album. Developers can sometimes alleviate these inconsistencies manually, by finding and ensuring proper ordering of writes. For example, the application can create the new album and wait for it to be replicated before posting on the wall. Then, concurrent access by Bob is not a problem, but a failure may leave behind an orphan album not linked to any user. The developer can deal with this problem by logging and replaying actions—which amounts to implementing rudimentary transactions—or garbage collecting dangling structures. This non-transactional approach places significant burden on developers.

We are not the first to point out the benefits of transactions to data center applications. Sinfonia uses transactions for infrastructure services [3, 4], while Percolator [38] uses them for search indexing. Both systems target applications on a single site, whereas we target geo-replicated applications that span many sites.

One way to provide transactions in a geo-replicated setting is to partition the data across several databases, where each database has its primary at a different site. The databases are replicated asynchronously across all sites, but each site is the primary for only one of the partitions. Unfortunately, with this solution, transactions cannot span multiple partitions, limiting their utility to applications.

Key features. Walter provides a unique combination of features to support geo-replicated web applications:

- *Asynchronous replication across sites.* Transactions are replicated lazily in the background, to reduce latency.
- *Efficient update-anywhere for certain objects.* Counting sets can be updated efficiently anywhere, while other objects can be updated efficiently at their preferred site.
- *Freedom from conflict-resolution logic,* which is complex and burdensome to developers.
- *Strong isolation within each site.* This is provided by the PSI property, which we cover below.

Existing systems do not provide some of the above features. For instance, eventually consistent systems such as [44, 47] require conflict-resolution logic; primary-copy database systems do not support any form of update-anywhere. We discuss related work in more detail in Section 9.

Overview of PSI. Snapshot isolation [8] is a popular isolation condition provided by commercial database systems such as Oracle and SQLServer. Snapshot isolation ensures that (a) transactions read from a snapshot that reflects a single commit ordering of transactions, and (b) if two concurrent transactions have a write-write conflict, one must be aborted. By imposing a single commit ordering, snapshot isolation forces implementations to coordinate transactions on commit, even when there are no conflicts (Section 3.1).

Parallel snapshot isolation extends snapshot isolation by allowing different sites to have different commit orderings. For example, suppose site A executes transactions T_1, T_2 and site B executes transactions T_3, T_4. PSI allows site A to first incorporate just T_1, T_2 and later T_3, T_4, while site B first incorporates T_3, T_4 and later T_1, T_2. This flexibility is needed for asynchronous replication: site A (or site B) can commit transactions T_1, T_2 (or T_3, T_4) without coordinating with the other site and later propagate the updates.

Although PSI allows different commit orderings at different sites, it still preserves the property of snapshot isolation that committed transactions have no write-write conflicts, thereby avoiding the need for conflict resolution. Furthermore, PSI preserves causal ordering: if a transaction T_2 reads from T_1 then T_1 is ordered before T_2 at every site. We give a precise specification of PSI in Section 3.

We believe PSI provides strong guarantees that are well-suited for web applications. Intuitively, PSI provides snapshot isolation for all transactions executed within a single site. PSI's relaxation over snapshot isolation is acceptable for web applications where each user communicates with one site at a time and there is no need for a global ordering of all actions across all users. In a social networking application, Alice in site A may post a message at the same time as Bob in site B. Under PSI, Alice may see her message first before seeing Bob's message, and Bob sees the opposite ordering, which is reasonable since Alice and Bob post concurrently. As another example, in an auction application, PSI allows bids on different objects to be committed in different orders at different sites. (In contrast, snapshot isolation requires the same ordering at all sites.)

Such relaxation is acceptable since the auction application requires bid ordering on each object separately, not across all objects.

Avoiding conflicts efficiently. To avoid write-write conflicts across sites, and implement PSI, Walter uses two techniques.

- *Preferred sites.* Each object is assigned a *preferred site*, which is the site where writes to the object can be committed without checking other sites for write conflicts. Walter executes and commits a transaction quickly if all the objects that it modifies have a preferred site where the transaction executes. Objects can be updated at any site, not just the preferred site. In contrast, some database systems have the notion of a *primary site*, which is the only site that can update the data. This notion is more limiting than the notion of a preferred site. For instance, suppose objects O_1 and O_2 are both replicated at sites 1 and 2, but the primary of O_1 is site 1 while the primary of O_2 is site 2. A transaction executing on site 1 can *read* both objects (since they are both replicated at site 1), but because the primary of O_2 is not site 1, the transaction can *write* only O_1—which is limiting to applications. In practice, this limitation is even more severe because database systems assign primary sites at the granularity of the whole database, and therefore non-primary sites are entirely read-only.
- *Conflict-free counting set objects.* Sometimes an object is modified frequently from many sites and hence does not have a natural choice for a preferred site. We address this problem with counting set (cset) objects. Transactions in Walter support not just read and write operations, but also operations on csets. Csets have the desirable property that transactions concurrently accessing the cset object never generate write-write conflicts. A cset is similar to a multiset in that it keeps a count for each element. But, unlike a multiset, the count could be negative [25]. A cset supports an operation *add*(x) to add element x, which increments the counter of x in the cset; and an operation *rem*(x) to remove x, which decrements the counter of x. Because increment and decrement commute, *add* and *rem* also commute, and so operations never conflict.
  For example, a group of concurrent cset operations can be ordered as *add*(x), *add*(y), *rem*(x) at one site, and ordered as *rem*(x), *add*(x), *add*(y) at another site. Both reach the final state containing just y with count 1. Note that removing element x from an empty cset results in -1 copies of element x, which is an *anti-element*: later addition of x to the cset results in the empty cset.

3. PARALLEL SNAPSHOT ISOLATION

In this section, we precisely specify PSI—the guarantee provided by Walter—and we discuss its properties and implications. We start by reviewing snapshot isolation and explaining the framework that we use to specify properties (Section 3.1). Then, we give the exact specification of PSI and discuss its properties (Section 3.2). We next explain how to extend PSI to include set operations (Section 3.3). We then explain how developers can use PSI (Section 3.4) and csets (Section 3.5) to build their applications.

3.1 Snapshot isolation

We specify snapshot isolation by giving an abstract specification code that an implementation must emulate. The specification code is centralized to make it as simple as possible, whereas an implementation can be distributed, complex, and more efficient. An implementation code satisfies the specification code if both codes produce the same output given the same input (e.g., [32]). The input

```
operation startTx(x)
  x.startTs ← new monotonic timestamp
  return OK

operation write(x, oid, data)
  append ⟨oid, DATA(data)⟩ to x.updates
  return OK

operation read(x, oid)
  return state of oid from x.updates and Log up to timestamp x.startTs

operation commitTx(x)
  x.commitTs ← new monotonic timestamp
  x.status ← chooseOutcome(x)
  if x.status = COMMITTED
  then append x.updates to Log with timestamp x.commitTs
  return x.status
```

Figure 1: Specification of snapshot isolation.

```
function chooseOutcome(x)
  if some write-conflicting transaction has committed after x started
  then return ABORTED
  else if some write-conflicting transaction has aborted after x started
       or is currently executing
  then return (either ABORTED or COMMITTED)    // non-deterministic choice
  else return COMMITTED
```

Figure 2: Transaction outcome in snapshot isolation.

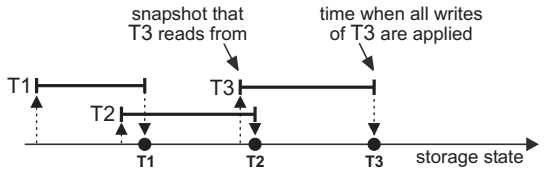


Figure 3: Depiction of snapshot isolation. The writes of T_1 are seen by T_3 but not T_2 as T_2 reads from a snapshot prior to T_1's commit.

is given by calls to operations to start a transaction, read or write data, commit a transaction, etc. The output is the return value of these operations. Many clients may call the operations of the specification concurrently, resulting possibly in many outstanding calls; however, the body of each operation is executed one at a time, using a single thread.

The specification is given in Figures 1 and 2 and depicted in Figure 3. It is assumed that clients start a transaction x with x initially $\perp$, then perform a sequence of reads and/or writes, and then try to commit the transaction. The behavior is unspecified if any client fails to follow this discipline, say by writing to a transaction that was never started. To start transaction x, the code obtains a new monotonically increasing timestamp, called the *start timestamp* of x. The timestamp is stored as an attribute of x; in the code, x is passed by reference. To write an object in transaction x, the code stores the object id and data in a temporary update buffer. To read an object, the code uses the update buffer—to check for any updates to the object written by the transaction itself—as well as a snapshot of the state when the transaction began. To determine the snapshot, the code maintains a *Log* variable with a sequence of object ids, data, and timestamps for the writes of previously-committed transactions. Only committed transactions are in the log, not outstanding ones. A read of an object reflects the updates in *Log* up to the transaction's start timestamp. To commit transaction x, the code obtains a new monotonically increasing timestamp, called the *commit timestamp* of x. It then determines the outcome of a transaction according to the function in Figure 2. This function indicates the cases

when the outcome is abort, commit, or either one chosen nondeterministically.[1] The code considers what happens after x started: if some write-conflicting transaction committed then the outcome is abort, where a *write-conflicting transaction* is one that writes an object that x also writes. Otherwise if some write-conflicting transaction has aborted or is currently executing—meaning it has started but its outcome has not been chosen—then the outcome is either abort or commit, chosen nondeterministically. Otherwise, the outcome is commit. If the outcome is commit, the writes of x are appended to *Log* with x's commit timestamp.

Note that the specification keeps internal variables—such as the log, timestamps, and other attributes of a transaction—but an implementation need not have these variables. It needs to emulate only the return values of each operation.

The above specification of snapshot isolation implies that any implementation must satisfy two key properties [51, Page 362]:

SI PROPERTY 1. *(Snapshot Read) All operations read the most recent committed version as of the time when the transaction began.*

SI PROPERTY 2. *(No Write-Write Conflicts) The write sets of each pair of committed concurrent transactions must be disjoint.*

Here, we say that two committed transactions are *concurrent* if one of them has a commit timestamp between the start and commit timestamp of the other.

Snapshot isolation is inadequate for a system replicated at many sites, due to two issues. First, to define snapshots, snapshot isolation imposes a total ordering of the commit time of all transactions, even those that do not conflict[2]. Establishing such an ordering when transactions execute at different sites is inefficient. Second, the writes of a committed transaction must be immediately visible to later transactions. Therefore a transaction can commit only after its writes have been propagated to all remote replicas, thereby precluding asynchronous propagation of its updates.[3] We define PSI to address these problems.

3.2 Specification of PSI

We define PSI as a relaxation of snapshot isolation so that transactions can propagate asynchronously and be ordered differently across sites. Note that the PSI specification does not refer to preferred sites, since they are relevant only to the implementation of PSI. The specification code is given in Figures 4 and 5 and depicted in Figure 6. As before, the specification is abstract and centralized—there is a single thread that executes the code without interleaving—but we expect that implementations will be distributed. Each transaction x has a site attribute denoted *site*(x). There is a log per site, kept in a vector *Log* indexed by sites. A transaction has one commit timestamp per site. A transaction first commits locally, by writing its updates to the log at its site; subsequently, the transaction propagates to and commits at the remote sites. This propagation is performed by the **upon** statement which, at some non-deterministic time, picks a committed transaction x and a site s to which x has not been propagated yet, and then writes the updates of x to the log at s. (For the moment, we ignore the second line of the upon statement in the code.) As Figure 5 shows,

[1] Nondeterminism in specifications allows implementations to have either behavior.

[2] For example, suppose $A{=}B{=}0$ initially and transaction T_1 writes $A{\leftarrow}1$, transaction T_2 writes $B{\leftarrow}1$, and both commit concurrently. Then T_1 and T_2 do not conflict and can be ordered arbitrarily, so either $(A{=}1, B{=}0)$ or $(A{=}0, B{=}1)$ are valid snapshots for transactions to read. However, it is illegal for both snapshots to occur, because snapshot isolation either orders T_1 before T_2 or vice versa.

[3] A variant called weak snapshot isolation [15] allows a transaction to remain invisible to others even after it commits, but that does not address the first issue above.

```
operation startTx(x)
 x.startTs ← new monotonic timestamp
 return OK

operation write(x, oid, data)
 append ⟨oid, DATA(data)⟩ to x.updates
 return OK

operation read(x, oid)
 return state of oid from x.updates and Log[site(x)] up to timestamp x.startTs

operation commitTx(x)
 x.commitTs[site(x)] ← new monotonic timestamp
 x.status ← chooseOutcome(x)
 if x.outcome = COMMITTED
   append x.updates to Log[site(x)] with timestamp x.commitTs[site(x)]
 return x.status

upon [∃x, s: x.status = COMMITTED and x.commitTs[s] = ⊥ and
       ∀y such that y.commitTs[site(x)] < x.startTs : y.commitTs[s] ≠ ⊥]
 x.commitTs[s] ← new monotonic timestamp
 append x.updates to Log[s] with timestamp x.commitTs[s]
```

Figure 4: Specification of PSI.

```
function chooseOutcome(x)
 if some write-conflicting transaction has committed at site(x) after x started
   or is currently propagating to site(x)   // text has definition of "propagating"
 then return ABORTED
 else if some write-conflicting transaction has aborted after x started
       or is currently executing
 then return (either ABORTED or COMMITTED)
 else return COMMITTED
```

Figure 5: Transaction outcome in PSI.

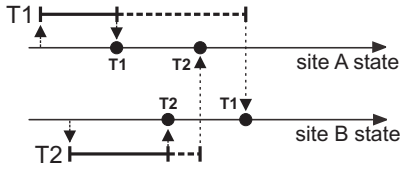


Figure 6: PSI allows a transaction to have different commit times at different sites. At site A, committed transactions are ordered as T1, T2. Site B orders them differently as T2, T1.

a transaction is aborted if there is some write-conflicting transaction that has committed at *site*(x) after x started or that is currently *propagating* to *site*(x); a transaction y is propagating to a site s if its status is committed but it has not yet committed at site s—that is, $y.status{=}$COMMITTED and $y.commitTs[s]{=}\perp$. Otherwise, if there is some concurrent write-conflicting transaction that has not committed, the outcome can be abort or commit. Otherwise, the outcome is commit. The outcome of a transaction is decided only once: if it commits at its site, the transaction is not aborted at the other sites. In Section 5.7, we discuss what to do when a site fails.

The above specification contains code that may be expensive to implement directly, such as monotonic timestamps and checks for write conflicts of transactions in different sites. We later give a distributed implementation that can avoid these inefficiencies.

From the specification, it can be seen that PSI replaces property 1 of snapshot isolation with the following:

PSI PROPERTY 1. *(Site Snapshot Read) All operations read the most recent committed version at the transaction's site as of the time when the transaction began.*

Intuitively, a transaction reads from a snapshot established at its site. In addition, PSI essentially preserves property 2 of snapshot

operation *setAdd*(x, *setid*, *id*)
 append $\langle$*setid*, ADD(*id*)$\rangle$ to x.*updates*
 return OK

operation *setDel*(x, *setid*, *id*)
 append $\langle$*setid*, DEL(*id*)$\rangle$ to x.*updates*
 return OK

operation *setRead*(x, *setid*)
 return state of *setid* from x.*updates* and *Log*[*site*(x)] up to timestamp x.*startTs*

Figure 7: Set operations in PSI specification.

isolation. To state the exact property, we say two transactions T_1 and T_2 are *concurrent at site s* if one of them has a commit timestamp at s between the start and commit timestamp of the other at s. We say the transactions are *somewhere-concurrent* if they are concurrent at *site*(T_1) or at *site*(T_2).

PSI PROPERTY 2. *(No Write-Write Conflicts) The write sets of each pair of committed somewhere-concurrent transactions must be disjoint.*

This property prevents the lost update anomaly (Section 3.4). The specification of PSI also ensures causal ordering:

PSI PROPERTY 3. *(Commit Causality Across Sites) If a transaction T_1 commits at a site A before a transaction T_2 starts at site A, then T_1 cannot commit after T_2 at any site.*

This property is ensured by the second line of the **upon** statement in Figure 4: x can propagate to a site s only if all transactions that committed at x's site before x started have already propagated to s. The property prevents a transaction x from committing before y at a remote site when x has observed the updates of y. The property also implies that write-conflicting transactions are committed in the same order at all sites, to prevent the state at different sites from diverging permanently.

3.3 PSI with cset objects

In the specification of PSI in Section 3.2, transactions operate on objects via read and write operations, but it is possible to extend the specification to support objects with other operations. We give the extension for cset objects, but this extension should apply to any object with commutative operations. To add an element to a cset, the code appends an entry $\langle$*setid*, ADD, *id*$\rangle$ to the transaction's update buffer (x.*updates*) and, on commit, appends this entry to the log. Similarly, to remove an element from a cset, the code appends entry $\langle$*setid*, DEL, *id*$\rangle$. To read a cset, the code computes the state of the cset: for each element, it sums the number of ADD minus the number of DEL in the log and the update buffer, thus obtaining a count for each element. Only elements with a non-zero count are returned by the read operation. Because the operations to add and remove elements in a cset commute, these operations do not cause a write conflict. Note that a cset object does not support a write operation since it does not commute with ADD. Figure 7 shows the code of the specification.

A cset may have many elements, and reading the entire cset could return large amounts of data. It is easy to extend the specification with an operation *setReadId* to return the count of a chosen element on a cset, by simply computing the state of the cset (using the log) to extract the count of that element.

3.4 Using PSI

One way to understand an isolation property is to understand what type of anomalous behavior it allows, so that developers know what to expect. In this section, we consider PSI from that standpoint, and we compare it against snapshot isolation and serializability. It is well-known that the weaker a property is, the more anomalous behaviors it has, but at the same time, the more efficiently it can be implemented. The anomalies allowed by PSI can be seen as the price to pay for allowing asynchronous replication.

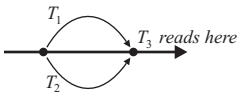
Anomaly	Serializability	Snapshot Isolation	PSI	Eventual Consistency
Dirty read	No	No	No	Yes
Non-repeatable read	No	No	No	Yes
Lost update	No	No	No	Yes
Short fork	No	Yes	Yes	Yes
Long fork	No	No	Yes	Yes
Conflicting fork	No	No	No	Yes

Dirty read. A transaction reads the update made by another transaction that has not yet committed; the other transaction may later abort or rewrite the object, making the data read by the first transaction invalid. *Example.* Initially $A{=}0$. T_1 writes $A{\leftarrow}1$ and $A{\leftarrow}2$ and commits; concurrently, T_2 reads $A{=}1$.

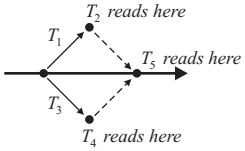
Non-repeatable read. A transaction reads the same object twice—once before and once after another transaction commits an update to it—obtaining different results. *Example.* Initially $A{=}0$. T_1 writes $A{\leftarrow}1$ and commits; concurrently T_2 reads $A{=}0$ and then reads $A{=}1$.

Lost update. Transactions make concurrent updates to some common object, causing one transaction to lose its updates. *Example.* Initially $A{=}0$. T_1 reads $A{=}0$, writes $A{\leftarrow}1$, and commits. Concurrently, T_2 reads $A{=}0$, writes $A{\leftarrow}2$, and commits.

Short fork. Transactions make concurrent disjoint updates causing the state to fork. After committing, the state is merged back. *Example.* Initially $A{=}B{=}0$. T_1 reads $A{=}B{=}0$, writes $A{\leftarrow}1$, and commits. Concurrently, T_2 reads $A{=}B{=}0$, writes $B{\leftarrow}1$, and commits. Subsequently, T_3 reads $A{=}B{=}1$.



Long fork. Transactions make concurrent disjoint updates causing the state to fork. After they commit, the state may remain forked but it is later merged back. *Example.* Initially $A{=}B{=}0$. T_1 reads $A{=}B{=}0$, writes $A{\leftarrow}1$, and commits; then T_2 reads $A{=}1, B{=}0$. T_3 and T_4 execute concurrently with T_1 and T_2, as follows. T_3 reads $A{=}B{=}0$, writes $B{\leftarrow}1$, and commits; then T_4 reads $A{=}0, B{=}1$. Finally, after $T_1, \ldots, T_4$ finish, T_5 reads $A{=}B{=}1$.



Conflicting fork. Transactions make concurrent conflicting updates causing the state to fork in a way that requires application-specific or ad-hoc rules to merge back. *Example.* Initially $A{=}0$. T_1 writes $A{\leftarrow}1$ and commits. Concurrently, T_2 writes $A{\leftarrow}2$ and commits. Some external logic determines that the value of A should be 3, and subsequently T_3 reads $A{=}3$.

Figure 8: Anomalies allowed by each isolation property.

Figure 8 shows various anomalies and whether each isolation property has those anomalies. Eventual consistency is very weak and allows all anomalies. The first three anomalies are well-known (e.g., [24]). Snapshot isolation and PSI prevent dirty and non-repeatable reads, because a transaction reads from a snapshot, and they prevent lost updates because there are no write-write conflicts. Snapshot isolation allows the state to fork, because two or more transactions may read from the same snapshot and make concurrent updates to different objects. We call this a *short fork*, also known as *write skew*, because the state merges after transactions commit. With PSI, the state may remain forked after transactions commit (when they execute in different sites), but the state is later merged when the transactions propagate across sites. Due to its longer du-

ration, we call this a *long fork*. A *conflicting fork* occurs when the states diverges due to conflicting updates, which is not allowed by PSI.

Long forks are acceptable in web applications when users in a site do not expect their updates to be instantly visible across all sites. If the user wants to know that her updates are visible everywhere, she can wait for her transaction to commit at all sites. In some cases, the fork may be noticeable to users: say, Alice posts a message on her social network wall saying that she is the first to flag a new promotion; she then confirms her statement by reading her friend's walls and seeing nothing there. With a long fork, Bob could be simultaneously doing the same thing from a different site, so that both Alice and Bob believe they posted their message first. One way to avoid possible confusion among users is for the application to show an "in-flight" mark on a freshly posted message; this mark is removed only when the message has been committed at all sites. Then, when Alice sees the mark, she can understand that her in-flight message may not yet be visible to all her friends.

Having discussed the anomalies of PSI, we now discuss ways that an application can use and benefit from PSI.

Multi-object atomic updates. With PSI, updates of a transaction occur together, so an application can use a transaction to modify many objects without exposing partial updates on each object.

Snapshots. With PSI, a transaction reads from a fixed consistent snapshot, so an application can use a transaction to ensure that it is reading consistent versions of different objects.

Read-modify-write operations. Because PSI disallows write-write conflicts, a transaction can implement any atomic read-modify-write operation, which reads an object and writes a new value based on the value read. Such operations include atomic increment and decrement of counters, atomic appends, and atomic edits.

Conditional writes. A particularly useful type of read-modify-write operation is a conditional write, which writes an object only if its content or version matches a value provided by the application. With PSI, this is performed by reading the object, evaluating the condition and, if it is satisfied, writing the object. This scheme can be extended to check and write many objects at once.

3.5 Using cset operations

A cset is a mapping from ids to counts, possibly negative. The mapping indicates how many times the element with a given id appears in the cset. There are two ways to use csets. First, when the count is useful to the application, a cset can be used as is. For example, a cset can keep the number of items in a shopping cart or inventory, the number of accesses to a data item, or the number of references to an object.

The second way to use a cset is as a conventional set, by hiding the counts from the user. For example, a cset can keep a list of friends, messages, active users, or photo albums. In these cases, the count has no meaning to the user. The application should be designed to keep the counts of elements at zero or one: the application should not add an element to a cset when the element is already present, or remove an element from a cset when the element is not there. In some cases, however, concurrent updates may cause the count to raise above one or drop below zero. For example, a user may add the same friend to her friends list, and do so concurrently at two different sites: the application sees a count of zero in both sites, and so it adds the friend once at each site. This situation is rare, because there must be updates to the *same* element in the *same* cset, and those updates must be concurrent, but it may happen. This is addressed by treating a count of one or more as present in the set, and count of zero or less as absent from the set. For example, when showing the list to the user, friends with negative counts are excluded. When the user adds a friend, if the count is negative, the application adds the friend enough times for the count to be one. When removing a friend, the application removes her enough times for the count to be zero. This is done by the application, transparently to the user.

4. SERVICE

This section describes how clients view and use Walter. Each site contains a Walter server and one or more application clients. Walter stores key-value object pairs grouped in containers (Section 4.1), where each container is replicated across multiple sites. The Walter client interface is exposed as a user-level library with functions to start transactions, read and write data, and commit transactions (Section 4.2). Walter provides fault tolerance by replicating data across sites (Section 4.3), and it allows users to trade-off durability for availability (Section 4.4).

4.1 Objects and containers

Walter stores objects, where an object has a key and a value. There are two types of objects: regular and cset. In a regular object, the value is an uninterpreted byte sequence, while in cset object, the value is a cset.

Each object is stored in a *container*, a logical organization unit that groups objects with some common purpose. For example, in a Web application, each user could have a container that holds all of her objects. To reduce space overhead, all objects in a container have the same preferred site, and Walter stores this information only once, as an attribute of the container. Administrators choose the preferred site to be the site most likely to modify the objects. For example, each user may have a designated site where she logs into the system (if she tries to log into a different site, she is redirected), and this would be the preferred site of her objects.

Object ids consist of a container id and a local id. The container id indicates to which container the object belongs, and the local id differentiates objects within a container. Since the container id is part of the object id, the container of an object cannot be changed.

4.2 Interface

Walter provides a client library for starting a transaction, manipulating objects, and committing a transaction, with the PSI semantics and operations explained in Sections 3.2 and 3.3. For regular objects, the available operations are read and write; for cset objects, the available operations are read, add element, and delete element.

Walter replicates transactions asynchronously, and the interface allows a client to receive a callback when (a) the transaction is disaster-safe durable (Section 4.4), and (b) the transaction is globally visible, meaning it has been committed at all sites.

4.3 Replication

Walter provides both durability and availability by replicating data within a single site and across multiple sites. Replication is transparent to clients: all the replicas of an object have the same object id, and the system accesses the replica closest to the client. An object need not be replicated at all sites and clients can read objects even if they are not replicated at the local site, in which case Walter fetches the data from a remote site.[4] A transaction commits

[4]In the PSI specification, data is replicated at every site, but an implementation need not do that, as long as it behaves identically in terms of responses to operations.

at every site, even where it is not replicated, following the semantics of PSI in Section 3.2: once a transaction is committed at a site, reads from that site see the effects of the transaction. Administrators choose how many replicas and where they are. These settings are stored as attributes of a container, so all objects of a container are replicated similarly.

4.4 Durability and availability

Walter provides two levels of durability:

(Normal Durability) When a transaction commits at its site, writes have been logged to a replicated cluster storage system [21, 28, 40, 48], so writes are not lost due to power failures. Data may be lost if an entire data center is wiped out by a disaster.

(Disaster-safe Durability) A transaction is considered *disaster-safe durable* if its writes have been logged at $f+1$ sites, where parameter f determines the desired fault tolerance level: up to f sites may fail without causing data loss. The default value of f is 1.

If an entire site s fails temporarily or is unreachable due to cross-site network issues, it may have transactions that were locally committed but not yet propagated to other sites. In that case, the application has two choices:

(Conservative) Wait for the site s to come back online, so that it can propagate the missing transactions. But then clients cannot write to objects whose preferred site is s until s comes back online—a loss of availability for some writes.

(Aggressive) Sacrifice the durability of a few committed transactions at site s for better availability, by replacing site s and abandoning its non-propagated transactions. Technically, this choice violates PSI, but one could extend the PSI definition to allow for lost committed transactions when a site fails or disconnects. Applications can wait for important transactions to be marked disaster-safe durable before confirming them to users.

Availability within a site comes from the availability of the cluster storage system: if the Walter server at a site fails, the system starts a new server, which can access the same cluster storage system. Availability under network partitions or disasters comes from cross-site replication. If a site fails, an application can warn users before they are redirected to another site, because users may see a different system state at the new site due to the semantics of PSI. In practice, the state at different sites diverges by only a few seconds.

5. DESIGN AND ALGORITHMS

This section describes Walter's design, emphasizing the protocols for executing and committing transactions. We first give an overview of the basic architecture (Section 5.1) and object versioning (Section 5.2). We then explain how to execute transactions (Section 5.3) and how to commit certain common transactions quickly (Section 5.4). Next, we explain how to commit other transactions (Section 5.5) and how transactions are replicated asynchronously (Section 5.6). Lastly, we consider failure recovery (Section 5.7) and scalability (Section 5.8).

5.1 Basic architecture

There are multiple sites numbered $1, 2, \ldots$ Each site contains a local Walter server and a set of clients. A client communicates with the server via remote procedure calls implemented by the API library. The server executes the actual operations to start and commit transactions, and to access objects.

Walter employs a separate *configuration service* to keep track of the currently active sites, and the preferred site and replica set for each object container. The configuration service tolerates failures by running as a Paxos-based state machine replicated across multiple sites. A Walter server confirms its role in the system by obtaining a lease from the configuration service, similar to what is done in [12, 46]. The lease assigns a set of containers to a preferred site, and it is held by the Walter server at that site. A Walter server caches the mapping from a container to its replica sites to avoid contacting the configuration service at each access. Incorrect cache entries do not affect correctness because a server rejects requests for which it does not hold the corresponding preferred site lease.

At $Server_i$: // i denotes the site number
- $CurrSeqNo_i$: integer with last assigned local sequence number
- $CommittedVTS_i$: vector indicating for each site how many transactions of that site have been committed at site i
- $History_i[oid]$: a sequence of updates of the form $\langle data, version\rangle$ to *oid*, where $version = \langle j{:}n\rangle$ for some j, n
- $GotVTS_i$: vector indicating for each site how many transactions of that site have been received by site i

Figure 9: Variables at server on each site.

5.2 Versions and vector timestamps

The PSI specification is centralized and uses a monotonic timestamp when a transaction starts and commits. But monotonic timestamps are expensive to produce across multiple sites. Thus, to implement PSI, Walter replaces them with version numbers and vector timestamps. A version number (or simply *version*) is a pair $\langle site, seqno\rangle$ assigned to a transaction when it commits; it has the site where the transaction executed, and a sequence number local to that site. The sequence number orders all transactions within a site. A vector timestamp represents a snapshot; it contains a sequence number for each site, indicating how many transactions of that site are reflected in the snapshot. A transaction is assigned a vector timestamp *startVTS* when it starts. For example, if $startVTS = \langle 2, 4, 5\rangle$ then the transaction reads from the snapshot containing 2 transactions from site 1, 4 from site 2, and 5 from site 3.

Given a version $v{=}\langle site, seqno\rangle$ and a vector timestamp *startVTS*, we say that v is *visible* to *startVTS* if $seqno \leq startVTS[site]$. Intuitively, the snapshot of *startVTS* has enough transactions from *site* to incorporate version v.

Figure 9 shows the variables at the server at site i. Variable $CurrSeqNo_i$ has the last sequence number assigned by the server, and $CommittedVTS_i[j]$ has the sequence number of the last transaction from each site j that was committed at site i. We discuss $History_i$ and $GotVTS_i$ in Sections 5.3 and 5.6.

5.3 Executing transactions

To execute transactions, the server at each site i maintains a history denoted $History_i[oid]$ with a sequence of writes/updates for each object *oid*, where each update is tagged with the version of the responsible transaction. This history variable is similar to variable *Log* in the PSI specification, except that it keeps a list per object, and it has versions not timestamps. When a transaction x starts, Walter obtains a new start vector timestamp *startVTS* containing the sequence number of the latest transactions from each site that were committed at the local site. To write an object, add to a cset, or remove from a cset, Walter stores this update in a temporary buffer $x.updates$. To read an object, Walter retrieves its state from the snapshot determined by *startVTS* and any updates in $x.updates$. Specifically, for a regular object, Walter returns the last update in $x.updates$ or, if none, the last update in the history visible to *startVTS*. For a cset object, Walter computes its state by

```
At Server_i:                                          // i denotes the site number
operation startTx(x)
  x.tid ← unique transaction id
  x.startVTS ← CommittedVTS_i
  return OK

operation write(x, oid, data): add ⟨oid, DATA(data)⟩ to x.updates; return OK
operation setAdd(x, setid, id): add ⟨setid, ADD(id)⟩ to x.updates; return OK
operation setDel(x, setid, id): add ⟨setid, DEL(id)⟩ to x.updates; return OK

operation read(x, oid)
  if oid is locally replicated
  then return state of oid reflecting x.updates and
       all versions in History_i[oid] visible to x.startVTS
  else return state of oid reflecting x.updates,
       the versions in History_site(oid)[oid] visible to x.startVTS, and
       the versions in History_i[oid] visible to x.startVTS

operation setRead(x, setid): same as read(x, oid)
```

Figure 10: Executing transactions.

```
At Server_i:                                          // i denotes the site number
function unmodified(oid, VTS): true if oid unmodified since VTS
function update(updates, version)
  for each ⟨oid, X⟩ ∈ updates do add ⟨X, version⟩ to History_i[oid]

operation commitTx(x)
  x.writeset ← {oid : ⟨oid, DATA(∗)⟩ ∈ x.updates }        // ∗ is a wildcard
  if ∀oid ∈ x.writeset : site(oid) = i then return fastCommit(x)
  else return slowCommit(x)

function fastCommit(x)
  if ∀oid ∈ x.writeset : unmodified(oid, startVTS) and oid not locked then
    x.seqno ← ++CurrSeqNo_i          // vertical bar indicates atomic region
    update(x.updates, ⟨i, x.seqno⟩)
    wait until CommittedVTS_i[i] = x.seqno−1
    CommittedVTS_i[i] ← x.seqno
    x.outcome ← COMMITTED
    fork propagate(x)
  else x.outcome ← ABORTED
  return x.outcome
```

Figure 11: Fast commit.

applying the updates in the history visible to *startVTS* and the updates in $x.updates$.

The above explanation assumes an object is replicated locally. If not, its local history $History_i[oid]$ will not have all of the object's updates (but it may have some recent updates). Therefore, to read such an object, Walter retrieves the data from the object's preferred site and merges it with any updates in the local history and in $x.updates$. To write, Walter buffers the write in $x.updates$ and, upon commit, stores the update in the local history while it is being replicated to other sites; after that, the local history can be garbage collected. Figure 10 shows the detailed pseudocode executed by a server. Recall that clients invoke the operations at the local server using a remote procedure call (not shown). The code is multi-threaded and we assume that each line is executed atomically.

5.4 Fast commit

For transactions whose write-set has only objects with a local preferred site, Walter uses a fast commit protocol. The write-set of a transaction consists of all oids to which the transaction writes; it excludes updates to set objects. To fast commit a transaction x, Walter first determines if x can really commit. This involves two checks for conflicts. The first check is whether all objects in the write-set are unmodified since the transaction started. To perform this check, Walter uses the start vector timestamp: specifically, we say that an object *oid* is *unmodified since* $x.startVTS$ if all versions of *oid* in the history of the local site are visible to $x.startVTS$. The second check is whether all objects in the write-set of x are unlocked; intuitively, a locked object is one being committed by the slow commit protocol (Section 5.5). If either check fails, then x is aborted. Otherwise, Walter proceeds to commit x, as follows. It assigns a new local sequence number to x, and then applies x's updates to the histories of the modified objects. Walter then waits until the local transaction with preceding sequence number has been committed. This typically happens quickly, since sequence numbers are assigned in commit order. Finally, transaction x is marked as committed and Walter propagates x to remote sites asynchronously as described in Section 5.6. Figure 11 shows the detailed pseudocode. The notation $site(oid)$ denotes the preferred site of *oid*. As before, we assume that each line is executed atomically. A vertical bar indicates a block of code with multiple lines that is executed atomically.

```
At Server_i:                                          // i denotes the site number
function slowCommit(x)
  // run 2pc among preferred sites of updated objects
  sites ← {site(oid) : oid ∈ x.writeset}
  pfor each s ∈ sites do                              // pfor is a parallel for
    vote[s] ← remote call prepare(x.tid,
                {oid ∈ x.writeset : site(oid) = s}, x.startVTS)
  if ∀s ∈ sites : vote[s] = YES then
    x.seqno ← ++CurrSeqNo_i          // vertical bar indicates atomic region
    update(x.updates, ⟨i, x.seqno⟩)
    wait until CommittedVTS_i[i] = x.seqno − 1
    CommittedVTS_i[i] ← x.seqno
    release locks (at this server) with owner x.tid
    x.outcome ← COMMITTED
    fork propagate(x)
  else
    pfor each s ∈ sites such that vote[s] = YES do remote call abort(x.tid)
    x.outcome ← ABORTED
  return x.outcome

function prepare(tid, localWriteset, startVTS)
  if ∀oid ∈ localWriteset : oid not locked and unmodified(oid, startVTS) then
    for each oid ∈ localWriteset do lock oid with owner tid
    return YES
  else return NO

function abort(tid)
  release locks (at this server) with owner tid
```

Figure 12: Slow commit.

5.5 Slow commit

Transactions that write a regular object whose preferred site is not local must be committed using the slow commit protocol, which employs a type of two-phase commit among the preferred sites of the written objects (not across all replicas of the objects). The purpose of two-phase commit is to avoid conflicts with instances of fast commit and other instances of slow commit. To commit a transaction x, the server at the site of the transaction acts as the coordinator in the two-phase protocol. In the first phase, the coordinator asks the (servers at the) preferred sites of each written object to vote based on whether those objects are unmodified and unlocked. If an object is modified at the preferred site, then an instance of fast commit conflicts with x; if the object is locked at the preferred site, then another instance of slow commit conflicts with x. If either case occurs, the site votes "no", otherwise the site locks the objects and votes "yes". If any vote is "no", the coordinator tells the sites to release the previously acquired locks. Otherwise, the coordinator proceeds to commit x as in the fast commit protocol: it assigns a sequence number to x, applies x's updates to the object histories, marks x as committed, and propagates x asynchronously. When x

```
At Server_i:                                   // i denotes the site number
function propagate(x)
 send ⟨PROPAGATE, x⟩ to all servers
 wait until ∀oid∈x.writeset: received ⟨PROPAGATE-ACK, x.tid⟩
      from f+1 sites replicating oid including site(oid)
 mark x as disaster-safe durable
 send ⟨DS-DURABLE, x⟩ to all servers
 wait until received ⟨VISIBLE, x.tid⟩ from all sites
 mark x as globally visible

when received ⟨PROPAGATE, x⟩ from Server_j and
    GotVTS_i ≥ x.startVTS and GotVTS_i[j] = x.seqno−1 do
 if i ≠ j then update(items in x.updates replicated in this site, ⟨j : x.seqno⟩)
 // when i = j, update has been applied already when transaction committed
 GotVTS_i[j] = x.seqno
 send ⟨PROPAGATE-ACK, x.tid⟩ to Server_j

when received ⟨DS-DURABLE, x⟩ and ⟨PROPAGATE, x⟩ from Server_j and
    CommittedVTS_i ≥ x.startVTS and CommittedVTS_i[j] = x.seqno−1 do
 CommittedVTS_i[j] ← x.seqno
 release all locks with owner x.tid
 send ⟨VISIBLE, x.tid⟩ to Server_j
```

Figure 13: Transaction replication.

commits, a site releases the acquired locks when x is propagated to it. Figure 12 shows the detailed pseudocode.

5.6 Asynchronous propagation

After a transaction commits, it is propagated asynchronously to other sites. The propagation protocol is simple: the site of a transaction x first copies the objects modified by x to the sites where they are replicated. The site then waits until *sufficiently many sites* indicate that they received (a) transaction x, (b) all transactions that causally precede x according to $x.startVTS$, and (c) all transactions of x's site with a smaller sequence number. "Sufficiently many sites" means at least $f+1$ sites replicating each object including the object's preferred site, where f is the disaster-safe tolerance parameter (Section 4.4). At this point, x is marked as disaster-safe durable and all sites are notified. Transaction x commits at a remote site j when (a) site j learns that x is disaster-safe durable, (b) all transactions that causally precede x are committed at site j, and (c) all transactions of x's site with a smaller sequence number are committed at site j. When x has committed at all sites, it is marked as globally visible. The pseudocode is shown in Figure 13. Vector $GotVTS_i$ keeps track of how many transactions site i has received from each other site. Note that when a site i receives a remote transaction and updates the history of its objects, the transaction is not yet committed at i: it commits only when $CommittedVTS_i[j]$ is incremented. The code omits simple but important optimizations: when server i propagates transaction x to a remote server, it should not send all the updates of x, just those updates replicated at the remote server. Similarly, when it sends a DS-DURABLE message, a server need not include the updates of x again.

5.7 Handling failures

Recovering from client or server failure. If a client crashes, its outstanding transactions are aborted and any state kept for those transactions at the server is garbage collected. Each server at a site stores its transaction log in a replicated cluster storage system. When a Walter server fails, the replacement server resumes propagation for those committed transactions that have not yet been fully propagated.

Handling a site failure. An entire site s may fail due to a disaster or a power outage. Such failure is problematic because there may be committed transactions at s that were not yet replicated at other sites. As explained in Section 4.4, Walter offers two site recovery options: conservative and aggressive. Recall that the conservative option is to wait for s to come back online, while the aggressive option is to remove s and reassign the preferred site of its containers to another site. To remove a failed site, Walter uses the configuration service (Section 5.1). Each configuration indicates what sites are active. Before switching to a new configuration that excludes site s, the configuration service must find out the transactions committed by s that will survive across the configuration change. Transaction x of site s survives if x and all transactions that causally precede x and all transactions of s with a smaller sequence number have been copied to a site in the new configuration. The configuration service queries the sites in the new configuration to discover what transactions survive. Then, it asks each site to discard the replicated data of non-surviving transactions and, in the background, it completes the propagation of surviving transactions that are not yet fully replicated. Finally, the configuration service reassigns the preferred site of containers of s to another site, by having another site take over the appropriate leases. While reconfiguration is in progress, sites that are still active continue to commit transactions, except transactions that write to objects whose preferred site was s, which are postponed until those objects get a new preferred site.

Re-integrating a previously failed site. When a previously removed site s recovers, it must be re-integrated into the system. The configuration service starts a new reconfiguration that includes s. To switch to the new configuration, s must first synchronize with its replacement site s' to integrate modifications committed by s'. Once synchronization is finished, s takes over the lease for being the preferred site for the relevant containers, and the new configuration takes effect.

5.8 Scalability

Walter relies on a single server per site to execute and commit transactions, which can become a scalability bottleneck. A simple way to scale the system is to divide a data center into several "local sites", each with its own server, and then partition the objects across the local sites in the data center. This is possible because Walter supports partial replication *and* allows transactions to operate on an object not replicated at the site—in which case, the transaction accesses the object at another site within the same data center. We should note that PSI allows sites to diverge; to avoid exposing this divergence to users, applications can be designed so that a user always log into the same local site in the data center. Another approach to scalability, which we do not explore in this paper, is to employ several servers per site and replace the fast commit protocol of Section 5.4 with distributed commit.

6. IMPLEMENTATION

The Walter implementation has a client-side library and a server, written in C++, with a total of 30K lines of code. There is also a PHP interface for web development with 600 lines of code. The implementation differs from the design as follows. First, each Walter server uses direct-attached storage devices, instead of a cluster storage system. Second, we have not implemented the scheme to reintegrate a failed site (Section 5.7): currently, the administrator must invoke a script manually to do that. Third, the client interface, shown in Figure 14, differs cosmetically from the specification in Section 3.2, due to the specifics of C++ and PHP. In C++, there is a Transaction class and operations are methods of this class. Functions read, setRead, and setReadId return the data via a parameter (the C++ return value is a success indication). setRead provides an

Method	Description
void start() int commit() int abort()	start transaction try to commit abort
int read(Oid o, char **buf) int write(Oid o, char *buf, int len) Oid newid(ContainerId cid, OType otype)	read object write object get new oid
int setAdd(Oid cset, Id id) int setDel(Oid cset, Id id) int setRead(Oid cset, IdSetIterator **iter) int setReadId(Oid cset, Id id, int *answer)	add *id* to *cset* delete *id* from *cset* read *cset* read *id* in *cset*

C++ Example:

```
Tx x;
x.start();
len = x.read(o1, &buf);
err = x.write(o2, buf, len);
...
res = x.commit();
```

PHP Example:

```
$x = waStartTx();
$buf = waRead($x, $o1);
$err = waWrite($x, $o2, $buf);
...
$res = waCommit($x);
```

Figure 14: Basic C++ API for Walter and C++ and PHP examples.

iterator for the ids in a cset. setReadId indicates the count of an identifier in a cset. commit can optionally inform the client via supplied callbacks—not shown—when the transaction is disaster-safe durable and globally visible (i.e., committed at all sites). There is a function newid to return a fresh oid, explained below.

There are no specialized functions to create or destroy objects. Conceptually, all objects always exist and are initialized to *nil*, without any space allocated to them. If a client reads a never-written object, it obtains *nil*. Function *newid* returns a unique oid of a never-written object of a chosen type (regular or cset) in a chosen container. Destroying a regular object corresponds to writing *nil* to it, while destroying a cset object corresponds to updating its elements so that they have zero count. There are some additional functions (not shown), including (a) management functions for initialization, shutdown, creating containers, and destroying containers; and (b) functions that combine multiple operations in a single RPC to the server, to gain performance; these include functions for reading or writing many objects, and for reading all objects whose ids are in a cset. The functions to create and destroy containers run outside a transaction; we expect them to be used relatively rarely. Identifiers for containers and objects are currently restricted to a fixed length, but it would be easy to make them variable-length.

The server stores object histories in a persistent log and maintains an in-memory cache of recently-used objects. The persistent log is periodically garbage collected to remove old entries. The entries in the in-memory cache are evicted on an LRU basis. Since it is expensive to reconstruct csets from the log, the eviction policy prefers to evict regular objects rather than csets. There is an in-memory index that keeps, for each object, a list of updates to the object, ordered from most to least recent, where each update includes a pointer to the data in the persistent log and a flag of whether the data is in the cache. To speed up system startup and recovery, Walter periodically checkpoints the index to persistent storage; the checkpoint also describes transactions that are being replicated. Checkpointing is done in the background, so it does not block transaction processing. When the server starts, it reconstructs the index from the checkpointed state and the data in the log after the checkpoint.

To improve disk efficiency, Walter employs group commit to flush many commit records to disk at the same time. To reduce the number of threads, the implementation makes extensive use of asynchronous calls and callbacks when it invokes blocking and slow operations. To enhance network efficiency, Walter propagates transactions in periodic batches, where each batch remotely copies all transactions that committed since the last batch.

```
Tx x;
x.start();
x.read(oidA, &profileA);
x.read(oidB, &profileB);
(* continues in next column *)
```

```
x.setAdd(profileA.friendlist, oidB);
x.setAdd(profileB.friendlist, oidA);
success = x.commit();
```

Figure 15: Transaction for *befriend* operation in WaltSocial.

The protocol for slow commit may starve because of repeated conflicting instances of fast commit. A simple solution to this problem is to mark objects that caused the abort of slow commit and briefly delay access to them in subsequent fast commits: this delay would allow the next attempt of slow commit to succeed. We have not implemented this mechanism since none of our applications use slow commit.

7. APPLICATIONS

Using Walter, we built a social networking web site (WaltSocial) and ported a third-party Twitter-like application called *ReTwis* [2]. Our experience suggests that it is easy to develop applications using Walter and run them across multiple data centers.

WaltSocial. WaltSocial is a complete implementation of a simple social networking service, supporting the common operations found in a system such as Facebook. These include *befriend*, *status-update*, *post-message*, *read-info* as well as others. In WaltSocial, each user has a profile object for storing personal information (e.g., name, email, hobbies) and several cset objects: a *friend-list* has oids of the profile objects of friends, a *message-list* has oids of received messages, an *event-list* has oids of events in the user's activity history, and an *album-list* has oids of photo albums, where each photo album is itself a cset with the oids of photo objects.

WaltSocial uses transactions to access objects and maintain data integrity. For example, when users A and B *befriend* each other, a transaction adds A's profile oid to B's friend-list and vice versa (Figure 15). To *post-message* from A to B, a transaction writes an object m with the message contents and adds its oid to B's message-list and to A's event-list.

Each user has a container that stores her objects. The container is replicated at all sites to optimize for reads. The system directs a user to log into the preferred site of her container. User actions are confirmed when transactions commit locally.

ReTwis. ReTwis is a Twitter-clone written in PHP using the Redis key-value store [1]. Apart from simple get/put operations, this application makes extensive use of Redis's native support for certain atomic operations, such as adding to or removing from a list, and adding or subtracting from an integer. In Redis, cross-site replication is based on a master-slave scheme. For our port of ReTwis, we replace Redis with Walter, so that ReTwis can update data on multiple sites. We use Walter transactions and csets to provide the equivalent atomic integer and list operation in Redis.

For each user, ReTwis has a timeline that tracks messages posted by the users that the user is following. In the original implementation, a user's timeline is stored in a Redis list. When a user posts a message, ReTwis performs an atomic increment on a sequence number to generate a postID, stores the message under the postID, and appends the postID to each of her followers' timelines. When a user checks postings, ReTwis displays the 10 most recent messages from her timeline. To port ReTwis to use Walter, we make several changes: we use a cset object to represent each user's timeline so that different sites can add posts to a user's timeline without conflicts. To post a message, we use a transaction that writes a message

under a unique postID, and adds the postID to the timeline of every follower of the user.

We found the process of porting ReTwis to Walter to be quite simple and straightforward: a good programmer without previous Walter experience wrote the port in less than a day. Transactions allow the data structure manipulations built into Redis to be implemented by the application, while providing competitive performance (Section 8.7).

8. EVALUATION

We evaluate the performance of Walter and its applications (WaltSocial, ReTwis) using Amazon's EC2. The highlights of our results are the following:

- Transactions that modify objects at their preferred sites commit quickly, with a 99.9-percentile latency of 27ms on EC2. Committed transactions are asynchronously replicated to remote sites within twice the network round-trip latency.
- Transactions that modify csets outside of their preferred sites also commit quickly without cross-site coordination. WaltSocial uses csets extensively and processes user requests with a 99.9-percentile latency under 50ms.
- The overhead for supporting transactions in Walter is reasonable. ReTwis running on Walter has a throughput 25% smaller than running on Redis in a single site, but Walter allows ReTwis to scale to multiple sites.

8.1 Experimental setup

Unless stated otherwise, experiments run on Amazon's EC2 cloud platform. We use machines in four EC2 sites: Virginia (VA), California (CA), Ireland (IE), and Singapore (SG), with the following average round-trip latencies within and across sites (in ms):

	VA	CA	IE	SG
VA	0.5	82	87	261
CA		0.3	153	190
IE			0.5	277
SG				0.3

Within a site, the bandwidth between two hosts is over 600 Mbps; across sites, we found a bandwidth limit of 22 Mbps.

We use extra-large EC2 virtual machine instances, with 7 GB of RAM and 8 virtual cores, each equivalent to a 2.5 GHz Intel Xeon processor. Walter uses write-ahead logging, where commit logs are flushed to disk at commit time. Since one cannot disable write-caching at the disk on EC2, where indicated we run experiments on a private cluster outside of EC2, with machines with two quad core Intel Xeon E5520 2.27 GHz processors and 8 GB of RAM.

Each EC2 site has a Walter server, and we run experiments with different numbers of sites and replication levels, as shown below:

Experiment name	Sites	Replication level
1-site	VA	none
2-sites	VA, CA	2
3-sites	VA, CA, IE	3
4-sites	VA, CA, IE, SG	4

Our microbenchmark workload (Sections 8.2–8.5) consists of transactions that read or write a few randomly chosen 100-byte objects. (Changing the object size from 100 bytes to 1 KB yields similar results.) We choose to evaluate small transactions because our applications, WaltSocial and ReTwis, only access a few small objects in each transaction. We consider a transaction to be disaster-safe durable when it is committed at all sites in the experiment.

8.2 Base performance

We first evaluate the base performance of Walter, and compare it against Berkeley DB 11gR2 (BDB), a commercial open-source developer database library. The goal is to understand if Walter provides a usable base performance.

Benchmark setup. We configure BDB to use B-trees with default pagesize and snapshot isolation; parameters are chosen for the best performance. We configure BDB to have two replicas with asynchronous replication. Since BDB allows updates at only one replica (the primary), we set up the Walter experiment to also update at one site. To achieve good throughput in BDB, we must use many threads at the primary to achieve high concurrency. However, with many threads, EC2 machines perform noticeably worse than private machines. Therefore, we run the primary BDB replica in our private cluster (with write-caching at the disk enabled), and the other replica at the CA site of EC2. We do the same for Walter. Clients and the server run on separate hosts. For BDB, we use an RPC server to receive and execute client requests.

The workload consists of either read or write transactions each accessing one 100-byte object. We populate BDB and Walter with 50,000 keys, which fits in the 1 GB cache of both systems. Walter includes an optimization to reduce the number of RPCs, where the start and commit of each transaction are piggybacked onto the first and last access, respectively. Thus, transactions with one access require just one RPC in Walter and in BDB.

Results. Figure 16 shows that throughput of read and write transactions of Walter is comparable to that of BDB. Read throughput is CPU-bound and mainly limited by the performance of our RPC library in both systems. Walter's read throughput is slightly lower because it does more work than BDB by acquiring a local lock and assigning a start timestamp vector when a transaction starts. The commit and replication latency of BDB and Walter are also similar and not shown here (see Section 8.3 for Walter's latency).

8.3 Fast commit on regular objects

This microbenchmark evaluates the performance of transactions on regular objects, using fast commit.

Benchmark setup. The experiments involve one to four sites. Objects are replicated at all sites, and their preferred sites are assigned evenly across sites. At each site, we run multiple clients on different hosts to issue transactions as fast as possible to its local Walter server. There are several workloads: *read-only*, *write-only*, and *mixed*. Read-only or write-only transactions access one or five 100-byte objects. The mixed workload consists of 90% read-only transactions and 10% write-only transactions.

Result: throughput. Figure 17 shows Walter's aggregate throughput across sites as the number of sites varies. Read throughput is bounded by the RPC performance and scales linearly with the number of sites, reaching 157 Ktps (thousands of transactions per second) with 4 sites. Write throughput is lower than read throughput due to lock contention within a Walter server. Specifically, when a transaction commits, a thread needs to acquire a highly contended lock to check for transaction conflicts. Moreover, write throughput does not scale as well as read throughput as the number of

Name	Read Tx throughput	Write Tx throughput
Walter	72 Ktps	33.5 Ktps
Berkeley DB	80 Ktps	32 Ktps

Figure 16: Base read and write transaction throughput.

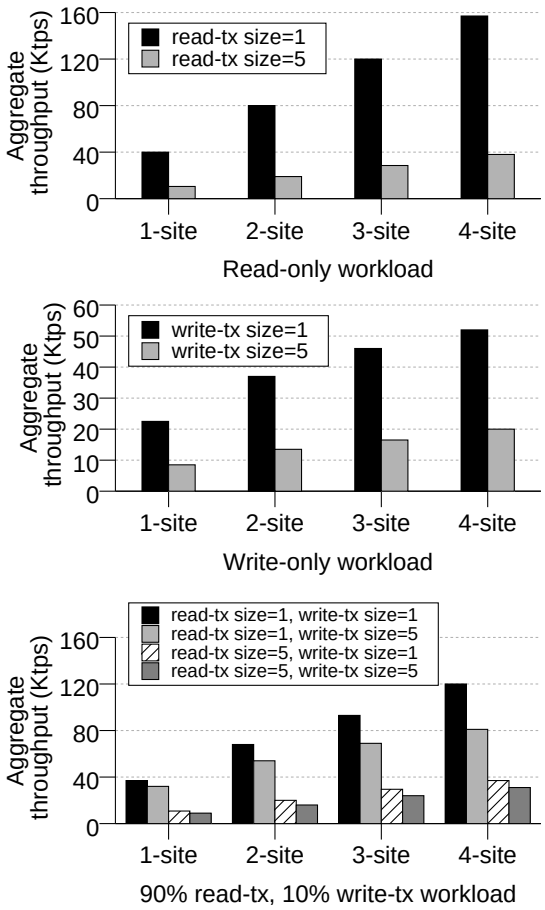


Figure 17: Aggregate transaction throughput on EC2.

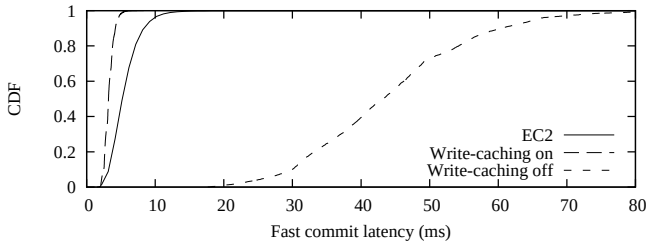


Figure 18: Fast commit latency on EC2 and our private cluster.

sites increases. This is because data is replicated at all sites, so the amount of work per write transaction grows with the number of sites. Yet, the cost of replication is lower than that of committing because replication is done in batches. Thus, the write throughput still grows with the number of sites, but not linearly. Note that the read and write throughput for transactions of size 1 in Figure 17 is only 50–60% of that in Figure 16 as a result of running this experiment on EC2 instead of the private cluster. In the mixed workload, performance is mostly determined by how many operations a transaction issues on average. For example, when there are 90% read-only transactions each reading one object and 10% write-only transactions each writing 5 objects, a transaction issues on average only 1.4 requests to the server. As a result, a relatively high aggregate throughput of 80 Ktps is reached across 4 sites.

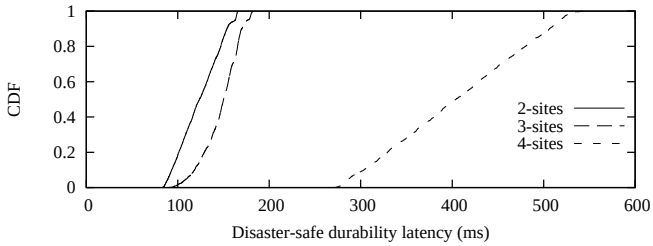


Figure 19: Replication latency for disaster-safe durability.

Result: latency. We measure the fast commit latency for write-only transactions accessing 5 objects. We record the time elapsed between issuing a commit and having the server acknowledge the commit completion. Figure 18 shows the latency distribution measured on EC2, and in our private cluster with and without write caching at the disk. The measurements were taken for a moderate workload in which clients issued enough requests to achieve 70% of maximal throughput. The points at the lower-end of the distributions in Figure 18 show latencies that we observe in a lightly loaded system.

Because there is no cross-site coordination, fast commit is quick: On EC2 the 99-percentile latency is 20 ms and the 99.9-percentile is 27 ms. Since the network latency within a site is low at 0.5 ms, the commit latency is dominated by the effects of queuing inside the Walter server and of flushing the commit log to disk when committing transactions at a high throughput. Figure 18 also shows the effect of disabling write-caching at the disk, measured on our private cluster. Even in that case, the 99.9-percentile latency of a fast commit is under 90 ms.

The latency for a committed transaction to become disaster-safe durable is dominated by the network latency across sites. As shown in Figure 19, the latency is distributed approximately uniformly between $[RTT_{max}, 2 * RTT_{max}]$ where RTT_{max} is the maximum round-trip latency between VA and the other three sites. This is because Walter propagates transactions in batches to maximize throughput, so a transaction must wait for the previous batch to finish.

The latency for a committed transaction to become globally visible is an additional RTT_{max} after it has become disaster-safe durable (not shown).

8.4 Fast commit on cset objects

We now evaluate transactions that modify csets.

Benchmark setup. We run the 4-site experiment in which each transaction modifies two 100-byte objects at the preferred site and adds an id to a cset with a remote preferred site.

Results. The latency distribution curve for committing transactions (not shown) is similar to the curve corresponding to EC2 in Figure 18. This is because transactions modifying csets commit via the same fast commit protocol as transactions modifying regular objects at their preferred site. Across 4 sites, the aggregate throughput is 26 Ktps, which is lower than the single-write transaction throughput of 52 Ktps shown in Figure 17. This is because the cset transactions issue 4 RPCs (instead of 1 RPC for the transactions in Figure 17), to write two objects, modify a cset, and commit.

8.5 Slow commit

We now evaluate the slow commit protocol for transactions modifying objects with different preferred sites. Unlike fast commit, slow commit requires cross-site coordination.

Benchmark setup. We run the 4-site experiments and have clients

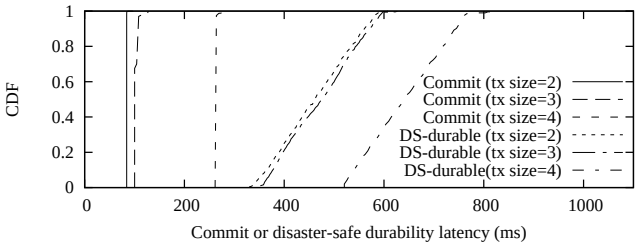


Figure 20: Latency of slow commit and replication.

issue write-only transactions at the VA site. We vary the size of a transaction from 2 to 4 objects. Each object written has a different preferred site: the first, second, third, and fourth object's preferred sites are VA, CA, IE, and SG respectively.

Results. Figure 20 shows the commit latency (left-most three lines) and the latency for achieving disaster-safe durability (right-most three lines). The commit latency is determined by the round-trip time between VA and the farthest preferred site of objects in the writeset. This is because slow commit runs a two-phase protocol among the preferred sites of the objects in the writeset. For example, for transactions of size 3, the commit latency is 87 ms, which is the round-trip time from VA to IE. The latency for disaster-safe durability is the commit latency plus the replication latency. The replication latency is the same as for fast commit: it is uniformly distributed between $[RTT_{max}, 2 * RTT_{max}]$, where RTT_{max} is the round-trip time between VA and SG.

To optimize performance, applications should minimize the use of slow commits. Both WaltSocial and ReTwis avoid slow commits by using csets.

8.6 WaltSocial performance

Transactions make it easy to develop WaltSocial. Our experiments also show that WaltSocial achieves good performance.

Workload setup. The WaltSocial experiments involve 4 sites in EC2. We populate Walter with $400,000$ users, each with 10 status updates and 10 wall postings from other users. We run many application clients at each site, where each client issues WaltSocial operations. An operation corresponds to a user action, and it is implemented by executing and committing a transaction that reads and/or writes several data objects (Section 7). We measure the latency and aggregate throughput for each operation. We also evaluate two mixed workloads: mix1 consists of 90% *read-info* operations and 10% update operations including *status-update*, *post-message* and *befriend*; mix2 contains 80% *read-info* operations and 20% update operations.

Operation throughput. Figure 21 shows the throughput in thousands operations per second (Kops/s) for each WaltSocial operation and for the mixed workloads. The *read-info* operation issues read-only transactions; it has the highest aggregate throughput at 40 Kops/s. The other operations issue transactions that update objects; their throughput varies from 16.5 Kops/s to 20 Kops/s, depending on the number of objects read and written in the transactions. The mixed workloads are dominated by *read-info* operations, hence their throughput values are closer to that of *read-info*. The achieved throughput is likely sufficient for small or medium social networks. To handle larger deployments, one might deploy several sites per data center to scale the system (Section 5.8) .

Operation latency. Figure 22 shows the latency of WaltSocial operations when the system has a moderate load. Operations finish

Operation	# objs+csets read	# objs written	# of csets written	Throughput (1000 ops/s)
read-info	3	0	0	40
befriend	2	0	2	20
status-update	1	2	2	18
post-message	2	2	2	16.5
mix1	2.9	0.5	0.3	34
mix2	2.8	0.7	0.5	32

Figure 21: Transaction size and throughput for Waltsocial operations.

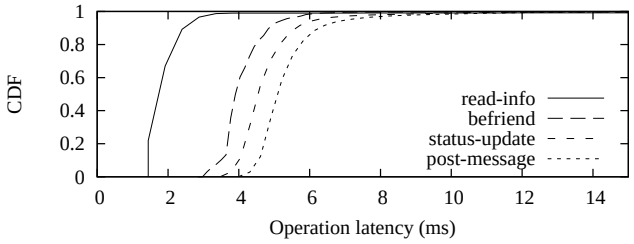


Figure 22: Latency of WaltSocial operations.

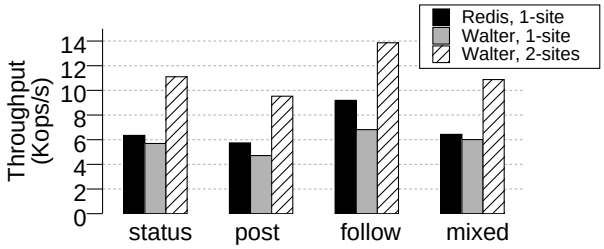


Figure 23: Throughput of ReTwis using Redis and Walter.

quickly because the underlying transactions involve no cross-site communication: transactions always read a local replica for any object and transactions that update data use cset objects. The 99.9-percentile latency of all operations in Figure 22 is below 50 ms. As each WaltSocial operation issues read/write requests to Walter in series, the latency is affected by the number of objects accessed by different WaltSocial operations. The *read-info* operation involves fewest objects and hence is faster than other operations.

8.7 ReTwis performance

We compare the performance of ReTwis using Walter and Redis as the storage system, to assess the cost of Walter.

Workload setup. The Walter experiments involve one or two sites. Redis does not allow updates from multiple sites, so the Redis experiments involve one site. Since Redis is a semi-persistent key-value store optimized for in-memory operations, we configure both Walter and Redis to commit writes to memory. We run multiple front-end web servers (Apache 2.2.14 with PHP 5.3.2) and client emulators at each site. We emulate $500,000$ users who issue requests to post a message (*post*), follow another user (*follow*), or read postings in their own timeline (*status*). The mixed workload consists of 85% *status*, 7.5% *post* and 7.5% *follow* operations.

Throughput comparison. Figure 23 shows the aggregate throughput (Kops/s) for different workloads when running ReTwis with Walter and Redis. As can be seen, with one site, ReTwis with Walter has similar performance as ReTwis with Redis: the slowdown is no more than 25%. For example, the throughput of the *post* operation for Walter (1 site) is 4713 ops/s, compared to 5740 ops/s for

Redis. But ReTwis with Walter can use multiple sites to scale the throughput. For example, the throughput of *post* using ReTwis with Walter on two sites is 9527 ops/s—twice the throughput of one site.

9. RELATED WORK

Transactions in data centers. Early transactional storage for data centers include Bigtable [12], Sinfonia [4], Percolator [38], and distributed B-trees [3]. Unlike Walter, these systems were designed for a single data center only.

Storage systems that span many data centers often do not provide transactions (e.g., Dynamo [16]), or support only restricted transactional semantics. For example, PNUTS [14] supports only one-record transactions. COPS [31] provides only read-only transactions. Megastore [7] partitions data and provides the ACID properties within a partition but, unlike Walter, it fails to provide full transactional semantics for reads across partitions.

Transactions in disconnected or wide-area systems. Perdis [19] is an object store with a check-out/check-in model for wide-area operations: it creates a local copy of remote data (check-out) and later reconciles local changes (check-in), relying on manual repair when necessary. For systems with mobile nodes, tentative update transactions [23] can commit at a disconnected node. Tentative commits may be aborted later due to conflicts when the hosts re-connect to servers, which requires reconciliation by an external user. In contrast to the above systems, Walter does not require burdensome operations for manual repair or reconciliation. Mariposa [45] is a wide-area system whose main focus is on incentivizing a site to run third-party *read-only queries*.

Database replication. There is much work on database replication, both commercially and academically. Commercial database systems support master-slave replication across sites: one site is the primary, the others are mirrors that are often read-only and updated asynchronously. When asynchronous mirrors are writable, applications must provide logic to resolve conflicts. On the academic side, the database replication literature is extensive; here we summarize relevant recent work. Replication schemes are classified on two axes [23]: (1) who initiates updates (primary-copy vs update-anywhere), and (2) when updates propagate (eager vs lazy). With *primary-copy*, objects have a master host and only the master initiates updates; with *update-anywhere*, any host may initiate updates. With *eager replication*, updates propagate to the replicas before commit; with *lazy replication*, replicas receive updates asynchronously after commit. All four combinations of these two dimensions are possible. Eager replication is implemented using distributed two-phase commit [9]. Later work considers primary-copy lazy replication and provides serializability by restricting the placement of each object's primary [13], or controlling when secondary nodes are updated [10, 36]. Update-anywhere lazy replication is problematic because conflicting transactions can commit concurrently at different replicas. Thus, recent work considers hybrids between eager and lazy replication: updates propagate after commit (lazy), but replicas also coordinate during transaction execution or commit to deal with conflicts (eager). This coordination may involve a global graph to control conflicts [6, 11], or atomic broadcast to order transactions [27, 37]. Later work considers snapshot isolation as a more efficient alternative to serializability [15, 17, 18, 30, 39, 52]. Walter differs from the above works because they ensure a stronger isolation property—serializability or snapshot isolation—which inherently requires coordination across sites to commit, whereas Walter commits common transactions without such coordination.

Federated transaction management considers techniques to execute transactions that span multiple database systems [41]. This work differs from Walter because it does not consider issues involving multiple sites and its main concern is to minimize changes to database systems, rather than avoiding coordination across sites.

Relaxed consistency. Some systems provide weaker consistency, where concurrent updates cause diverging versions that must be reconciled later by application-specific mechanisms [16, 34, 47]. *Eventual consistency* permits replicas to diverge but, if updates stop, replicas eventually converge again. Weak consistency may be tolerable [49], but it can lead to complex application logic. Inconsistency can also be quantified and bounded [5, 26, 54], to improve the user experience. Fork consistency [33] allows the observed operation history to fork and not converge again; it is intended for honest clients to detect the misbehavior of malicious servers rather than to provide efficient replication across sites.

Commutative data types. Prior work has shown how to exploit the semantics of data types to improve concurrency. In [50], abstract data types (such as sets, FIFO queues, and a bank account) are characterized using a table of commutativity relations where two operations conflict when they do not commute. In [20, 42], a lock compatibility table is used to serialize access to abstract data types, such as directory, set or FIFO queue, by exploiting the commutativity of their operations. Because these works aim to achieve serializability, not all operations on a set object are conflict-free (e.g., testing the membership of element a conflicts with the insertion of a in the set). As a result, operating on sets require coordination to check for potential conflicts. In contrast, since we aim to achieve the weaker PSI property, operations on Walter's cset objects are always free of conflicts, allowing each data center to read and modify these csets without any remote coordination.

Letia et al. [29] have proposed the use of commutative replicated data types to avoid concurrency control and conflict resolution in replicated systems. Their work has inspired our use of csets. Subsequent recent work [43] provides a theoretical treatment for such data types and others—which are together called conflict-free replicated data types or CRDTs—proposing sufficient conditions for replica convergence under a newly-defined strong eventual consistency model. While that work concerns replication of single operations/objects at a time, not transactions, one could imagine using general CRDTs with PSI and our protocols to replicate transactions efficiently. U-sets [43, 53] are a type of set in which commutativity is achieved by preventing a removed element from being added again. In contrast, csets achieve commutativity by augmenting elements with counts. Csets are similar to Z-relations [25], which are mappings from tuples to integers, used to allow for decidability of equivalence of queries in the context of query optimization.

Escrow transactions [35] update numeric data, such as account balances, by holding some amount in escrow to allow concurrent commutative updates. By exploiting commutativity, such transactions resemble transactions with csets, but they differ in two ways. First, escrow transactions operate on numeric data. Second, escrow transactions must coordinate among themselves to check the amounts in escrow, which does not serve our goal of avoiding coordination across distant sites.

10. CONCLUSION

Walter is a transactional geo-replicated key-value store with properties that make it appealing as the storage system for web ap-

plications. A key feature behind Walter is Parallel Snapshot Isolation (PSI), a precisely-stated isolation property that permits asynchronous replication across sites without the need for conflict resolution. Walter relies on techniques to avoid conflicts across sites, thereby allowing transactions to commit locally in a site. PSI thus permits an efficient implementation, while also providing strong guarantees to applications. We have demonstrated the usefulness of Walter by building a Facebook-like social networking application and porting a third-party Twitter clone. Both applications were simple to implement and achieved reasonable performance.

Acknowledgements

We are grateful to many people who helped us improve this work. Our shepherd Robbert van Renesse and the anonymous reviewers provided much useful feedback throughout the paper. Margo Seltzer made many suggestions on how to tune the performance of Berkeley DB. Frank Dabek, Wilson Hsieh, Frans Kaashoek, Christopher Mitchell, Rama Ramasubramanian, Mehul Shah, Chandramohan Thekkath, Michael Walfish, and Lidong Zhou provided several comments that helped us improve the presentation. This work was partially supported by NSF Award CNS-0720644.

References

[1] Redis: an open-source advanced key-value store. http://redis.io.

[2] A Twitter clone for the Redis key-value database. http://retwis.antirez.com.

[3] M. K. Aguilera, W. Golab, and M. A. Shah. A practical scalable distributed B-tree. In *International Conference on Very Large Data Bases*, pages 598–609, Aug. 2008.

[4] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. *ACM Transactions on Computer Systems*, 27(3):5:1–5:48, Nov. 2009.

[5] R. Alonso, D. Barbará, and H. Garcia-Molina. Data caching issues in an information retrieval system. *ACM Transactions on Database Systems*, 15(3):359–384, Sept. 1990.

[6] T. Anderson, Y. Breitbart, H. F. Korth, and A. Wool. Replication, consistency, and practicality: are these mutually exclusive? In *International Conference on Management of Data*, pages 484–495, June 1998.

[7] J. Baker et al. Megastore: Providing scalable, highly available storage for interactive services. In *5th Conference on Innovative Data Systems Research*, pages 223–234, Jan. 2011.

[8] H. Berenson et al. A critique of ANSI SQL isolation levels. In *International Conference on Management of Data*, pages 1–10, May 1995.

[9] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[10] Y. Breitbart et al. Update propagation protocols for replicated databases. In *International Conference on Management of Data*, pages 97–108, June 1999.

[11] Y. Breitbart and H. F. Korth. Replication and consistency in a distributed environment. *Journal of Computer and System Sciences*, 59(1):29–69, Aug. 1999.

[12] F. Chang et al. Bigtable: A distributed storage system for structured data. In *Symposium on Operating Systems Design and Implementation*, pages 205–218, Nov. 2006.

[13] P. Chundi, D. J. Rosenkrantz, and S. S. Ravi. Deferred updates and data placement in distributed databases. In *International Conference on Data Engineering*, pages 469–476, Feb. 1996.

[14] B. F. Cooper et al. PNUTS: Yahoo!'s hosted data serving platform. In *International Conference on Very Large Data Bases*, pages 1277–1288, Aug. 2008.

[15] K. Daudjee and K. Salem. Lazy database replication with snapshot isolation. In *International Conference on Very Large Data Bases*, pages 715–726, Sept. 2006.

[16] G. DeCandia et al. Dynamo: Amazon's highly available key-value store. In *ACM Symposium on Operating Systems Principles*, pages 205–220, Oct. 2007.

[17] S. Elnikety, S. Dropsho, and F. Pedone. Tashkent: Uniting durability with transaction ordering for high-performance scalable database replication. In *European Conference on Computer Systems*, pages 117–130, Apr. 2006.

[18] S. Elnikety, S. Dropsho, and W. Zwaenepoel. Tashkent+: Memory-aware load balancing and update filtering in replicated databases. In *European Conference on Computer Systems*, pages 399–412, Mar. 2007.

[19] P. Ferreira et al. Perdis: design, implementation, and use of a persistent distributed store. In *Recent Advances in Distributed Systems*, volume 1752 of *LNCS*, chapter 18. Springer-Verlag, Feb. 2000.

[20] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems*, 8(2):186–213, June 1983.

[21] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *ACM Symposium on Operating Systems Principles*, pages 29–43, Oct. 2003.

[22] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition tolerant web services. *ACM SIGACT News*, 33(2):51–59, June 2002.

[23] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *International Conference on Management of Data*, pages 173–182, June 1996.

[24] J. Gray and A. Reuter. *Transaction processing: concepts and techniques*. Morgan Kaufmann Publishers, 1993.

[25] T. J. Green, Z. G. Ives, and V. Tannen. Reconcilable differences. In *International Conference on Digital Telecommunications*, pages 212–224, Mar. 2009.

[26] H. Guo et al. Relaxed currency and consistency: How to say "good enough" in SQL. In *International Conference on Management of Data*, pages 815–826, June 2004.

[27] B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Transactions on Database Systems*, 25(3):333–379, Sept. 2000.

[28] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 84–92, Oct. 1996.

[29] M. Letia, N. Preguiça, and M. Shapiro. Consistency without concurrency control in large, dynamic systems. In *International Workshop on Large Scale Distributed Systems and Middleware*, Oct. 2009.

[30] Y. Lin, B. Kemme, M. P. no Martínez, and R. Jiménez-Peris. Middleware based data replication providing snapshot isolation. In *International Conference on Management of Data*, pages 419–430, June 2005.

[31] W. Lloyd, M. Freedman, M. Kaminsky, and D. Andersen. Don't settle for eventual: Stronger consistency for wide-area

storage with cops. In *ACM Symposium on Operating Systems Principles*, Oct. 2011.

[32] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[33] D. Mazières and D. Shasha. Building secure file systems out of byzantine storage. In *ACM Symposium on Principles of Distributed Computing*, pages 108–117, July 2002.

[34] L. B. Mummert, M. R. Eblig, and M. Satyanarayanan. Exploiting weak connectivity for mobile file access. In *ACM Symposium on Operating Systems Principles*, pages 143–155, Dec. 1995.

[35] P. E. O'Neil. The escrow transactional method. *ACM Transactions on Database Systems*, 11(4):405–430, Dec. 1986.

[36] E. Pacitti, P. Minet, and E. Simon. Fast algorithms for maintaining replica consistency in lazy master replicated databases. In *International Conference on Very Large Data Bases*, pages 126–137, Sept. 1999.

[37] M. Patino-Martinez, R. Jiménez-Peris, B. Kemme, and G. Alonso. MIDDLE-R: Consistent database replication at the middleware level. *ACM Transactions on Computer Systems*, 23(4):375–423, Nov. 2005.

[38] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Symposium on Operating Systems Design and Implementation*, pages 251–264, Oct. 2010.

[39] C. Plattner and G. Alonso. Ganymed: Scalable replication for transactional web applications. In *International Middleware Conference*, pages 155–174, Oct. 2004.

[40] Y. Saito et al. FAB: building distributed enterprise disk arrays from commodity components. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 48–58, Oct. 2004.

[41] R. Schenkel et al. Federated transaction management with snapshot isolation. In *Workshop on Foundations of Models and Languages for Data and Objects, Transactions and Database Dynamics*, pages 1–25, Sept. 1999.

[42] P. Schwarz and A. Spector. Synchronizing shared abstract types. *ACM Transactions on Computer Systems*, 2(3):223–250, Aug. 1984.

[43] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems*, Oct. 2011.

[44] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, and P. Maniatis. Zeno: Eventually consistent byzantine fault tolerance. In *Symposium on Networked Systems Design and Implementation*, pages 169–184, Apr. 2009.

[45] M. Stonebraker et al. Mariposa: a wide-area distributed database system. In *International Conference on Very Large Data Bases*, pages 48–63, Jan. 1996.

[46] J. Stribling, Y. Sovran, I. Zhang, X. Pretzer, J. Li, F. Kaashoek, and R. Morris. Simplifying wide-area application development with WheelFS. In *Symposium on Networked Systems Design and Implementation*, pages 43–58, Apr. 2009.

[47] D. B. Terry et al. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *ACM Symposium on Operating Systems Principles*, pages 172–183, Dec. 1995.

[48] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A scalable distributed file system. In *ACM Symposium on Operating Systems Principles*, pages 224–237, Oct. 1997.

[49] W. Vogels. Data access patterns in the amazon.com technology platform. In *International Conference on Very Large Data Bases*, page 1, Sept. 2007.

[50] W. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers*, 37(12):1488–1505, Dec. 1988.

[51] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2009.

[52] S. Wu and B. Kemme. Postgres-R(SI): Combining replica control with concurrency control based on snapshot isolation. In *International Conference on Data Engineering*, pages 422–433, Apr. 2005.

[53] G. T. J. Wuu and A. J. Bernstein. Efficient solutions to the replicated log and dictionart problems. In *ACM Symposium on Principles of Distributed Computing*, pages 233–242, Aug. 1984.

[54] H. Yu and A. Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Transactions on Computer Systems*, 20(3):239–282, Aug. 2002.