FW-KV: Improving Read Guarantees in PSI

Masoomeh Javidi Kishi
Lehigh University
USA
maj717@lehigh.edu

Roberto Palmieri
Lehigh University
USA
palmieri@lehigh.edu

ABSTRACT

We present FW-KV, a novel distributed transactional in-memory key-value store that guarantees the Parallel Snapshot Isolation (PSI) correctness level. FW-KV's primary goal is to allow its read-only transactions to access more up-to-date (fresher) versions of objects than Walter, the state-of-the-art implementation of PSI. FW-KV achieves that without assuming synchrony or a synchronized clock service. The improved level of freshness comes at no significant performance degradation, especially in low contention workloads, as assessed by our evaluation study including two standard OLTP benchmarks, YCSB and TPC-C. The performance gap between FW-KV and Walter is less than 5% in low contention scenarios, and less than 28% in high contention.

CCS CONCEPTS

• **Theory of computation → Concurrency**; • **Information systems → Database transaction processing**.

KEYWORDS

Transactions, Consistency, Snapshot Isolation

ACM Reference Format:
Masoomeh Javidi Kishi and Roberto Palmieri. 2021. FW-KV: Improving Read Guarantees in PSI. In *22nd International Middleware Conference (Middleware '21), December 6–10, 2021, Québec city, QC, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3464298.3476131

1 INTRODUCTION

Snapshot Isolation (SI) [3] is a widely adopted consistency level often used as a practical alternative to Serializability [4], the gold standard criterion for concurrency control implementations [18, 28]. Informally, one of the great advantages of SI is that a transaction should not abort even though the set of values read (we name it *reading snapshot*), and not written, during its execution has been overwritten by a concurrent transaction [3]. By leveraging this property, along with a multi-versioned data repository, major database engines [26, 29] provide SI concurrency control on a single node by defining the reading snapshot as all the versions available at the time a transaction starts. An immediate consequence of this design is that read-only transactions, which never modify the data repository, can execute without the chance of aborting.

In a centralized deployment, the reading snapshot is often determined by assuming that time is measured by a shared atomic counter that advances whenever any transaction starts or commits [14]. In distributed systems where off-the-shelf hardware is assumed, nodes do not share a synchronized clock and the communication among them is asynchronous, SI transactions cannot simply define an up-to-date reading snapshot at the time they start because of the absence of a shared notion of time among nodes.

Walter [28] is a distributed transactional system whose concurrency control implements a relaxed variant of SI, called Parallel Snapshot Isolation (or PSI). In PSI, the transaction reading snapshot can be arbitrarily outdated in order to deal with the aforementioned absence of shared clocks among nodes (other relaxations are overviewed in Section 6).

Walter logically assigns objects to so called *preferred nodes*. A preferred node always stores the latest version of an object. The object might also be replicated on other *non-preferred* nodes, which might not always have the latest version of objects. If a transaction begins on a node N and reads an object whose preferred node is N (we name such a transaction *local*), then its reading snapshot is guaranteed to be up-to-date. Otherwise, when a transaction begins on a non-preferred node or any other node (for brevity, in both these cases we refer to this transaction as *non-local*), the read operations can return an *outdated* object version.

Walter attempts to patch the above issue by using asynchronous messages, sent outside the transaction critical path, aimed at periodically updating the logical clock of other nodes, including the non-preferred ones. However, until asynchronous messages are received, non-local read-only transactions can still return arbitrarily old versions. Another side effect of this solution is that non-local update transactions will be repeatedly aborted until the above asynchronous messages are delivered.

In this paper we present FW-KV, a distributed concurrency control that uses logical (vector) clocks [20] to implement an enhanced version of Walter's concurrency control with the goal of improving data freshness for read-only transaction. FW-KV exploits the fact that a common behavior for transactions is accessing mostly local objects [5]. For the remaining accesses, Walter must adhere to a possibly old reading snapshot. FW-KV improves this scenario for read-only transactions. Every access to a new node made by a read-only transaction is guaranteed to observe the most recent and correct reading snapshot. The *only* case in which a read-only transaction is prevented from accessing the latest reading snapshot is when multiple accesses target objects stored on the same node.

A practical example in which FW-KV always returns the most recent reading snapshot is when we consider the two transaction profiles `Order-Status` and `Payment` of the well-known benchmark TPC-C benchmark [10]. The former queries the status of a customer's last order from a warehouse to retrieve information about related order lines. The latter processes the payment for the customer and modifies the balance of the warehouse where the order took place. The read-only transaction `Order-Status` can see the latest version of the accessed objects modified by `Payment` since the first access is to retrieve the warehouse, and the subsequent read operations are on objects that have been committed along with that warehouse, regardless of the preferred node of the warehouse.

The properties ensured by FW-KV fit the characteristics of modern social network applications. In fact, the traditional understanding of those applications is that reading arbitrarily old values is admissible for users. However, users nowadays increasingly demand more stringent ordering requirements among updates (e.g., social media news).

The capability of FW-KV to observe fresh reading snapshots comes at the expense of some performance overhead. As empirically shown in our evaluation study, when application workload is dominated by read-only transactions, a typical case in many real applications and services [2, 30], the overhead of FW-KV becomes negligible. Such cases represent the practical sweet spot for FW-KV, in which the great scalability and performance of PSI (and Walter) are preserved while read-only transactions read up-to-date values.

The major algorithmic challenge in achieving FW-KV's goals is to deal with the (fast) technique used by Walter to update nodes' logical clocks upon transaction commits. In fact, since Walter's transaction reading snapshot can be arbitrarily old, vector clocks are updated without synchronously propagating causal dependency with other transactions.

Unlike Walter, the reading snapshot of a read-only transaction in FW-KV is established during its execution by means of attempting to include the newest versions of an object stored by a node that has not been contacted so far by this transaction. FW-KV ensures that by efficiently tracking some (but not all) transaction dependency relations. Update transactions execute with similar guarantees as in Walter, although FW-KV still attempts to improve data freshness by deploying a technique, similar to the one used in SCORe [23], where the reading snapshot is defined upon the first read operation.

To assess performance of FW-KV, we implement a distributed key-value store with the FW-KV concurrency control at its core, and contrast its performance against Walter and a well-known baseline distributed transactional system, named 2PC-baseline. In 2PC-baseline, all transactions, including read-only, validate read keys to ensure correct and the most recent reading snapshot, and use the Two-Phase Commit protocol (2PC) to commit [4]. We use two OLTP benchmarks, YCSB [8] and TPC-C [10], to generate transactional workload.

Results show that FW-KV improves read-only transactions' data freshness with a performance penalty of less than 5% in case of low contention, and it goes up to 20% and 28% in YCSB and TPC-C, respectively, when contention increases. FW-KV's capability of reading fresher data than Walter allows its update transactions to abort up to 3 times less in case asynchronous messages propagation is delayed due to network congestion.

The paper makes the following contributions. The FW-KV improves upon Walter's concurrency control by:

- Increasing data freshness of read-only transactions;
- Reducing occurrence of long fork anomaly;
- Enhancing the performance robustness in terms of update transactions' abort rate, in case asynchronous messages are delayed due to network congestion.

2 OVERVIEW, ASSUMPTIONS AND PROPERTIES

2.1 System Model

FW-KV assumes a system made of a set of nodes that do not share neither memory nor a global clock. Nodes communicate through message passing over reliable asynchronous channels, meaning messages are guaranteed to be eventually delivered unless a crash happens at the sender or receiver node. There is no assumption on the speed and on the level of synchrony among nodes.

2.2 Data Organization

Every node N_i maintains shared objects (or keys) adhering to the key-value model [24]. The data repository is multiversioned, meaning each shared object keeps a list of previous versions. Each version stores the value and the commit vector clock of the transaction that produced the version. In FW-KV, every shared key can be stored in an arbitrary preferred site. For object reachability, FW-KV implements a local look-up function using consistent hashing, a commonly used technique to map keys to nodes. An object is local with respect to a node if it is stored on that node, otherwise it is remote.

To survive failures, FW-KV assumes each preferred site is highly available, meaning the site is expected to implement a replication technique to resist faults. For simplicity in the explanation of our distributed concurrency control, we do not account for replication. In literature, many efficient techniques have been proposed to address the problem of preserving availability of a site (often called shard) [15, 19].

2.3 Transaction model

We model transactions as a sequence of read and write operations on shared objects (or keys), preceded by a *begin* operation, and followed by a *commit* or *abort* operation. A client begins a transaction on the co-located node and the transactions can read/write data belonging to any node; no a-priori knowledge on the accessed keys is assumed.

Every transaction starts with a client submitting it to the system, and finishes its execution informing the client about its final outcome: commit or abort. Transactions that do not execute any write operation are called read-only, otherwise they are update transactions. Read-only transactions are expected to be identified by the programmer.

A transaction is *local* if it begins on a node N and all its read operations access objects whose preferred node is N itself. Otherwise the transaction is *non-local*.

In terms of consistency level, FW-KV preserves Parallel Snapshot Isolation as the original Walter [28] protocol.

2.4 Freshness Level of Reading Snapshot

FW-KV improves the level of freshness of read operations on the original Walter's distributed concurrency control. We define the level of freshness for a read operation OP issued by a transaction T

on a shared object o as the metric that quantifies the gap between the version returned by OP and the latest version of o available in the data store at the time OP is issued. A property known as real-time order [22], often used in the definition of consistency levels (e.g., Strict Serializability [9]), is an expression of the level of freshness for read operations that mandates the access to at least the latest version of an object available before the start time of a transaction. Preserving real-time order is costly in distributed settings due to the absence of a shared notion of time [18, 22].

In FW-KV, the level of freshness depends on whether the transaction issuing the request is read-only or update.

Let us assume a read-only transaction T_{RO} executing on N_{RO} and issuing a read operation OP_i to access object o stored on node N_o. Operation OP_i returns the latest version of o if it is the first time for T_{RO} to access an object stored in N_o. After that, subsequent operations OP_j of T_{RO}, needing to contact N_o to read either o or any other object q stored in N_o, will return a version consistent with the version of o previously read, which might or might not match the latest commit of q. That means, if all read operations of T_{RO} access either local objects or objects stored on different nodes, the reading snapshot of T_{RO} is guaranteed to be the freshest possible (equivalent to guaranteeing real-time order).

An update transaction in FW-KV is guaranteed to return the latest version of its first read operation. After that, the logical clock associated with the node handling the first read is used to derive the versions to be returned by subsequent reads, regardless of the contacted nodes.

3 BACKGROUND & MOTIVATION

3.1 Walter & PSI

Walter [28] is a multi-version transactional key-value store that provides a relaxed version of SI called Parallel Snapshot Isolation (PSI). Walter uses a technique named *preferred site* where each object is logically assigned to a specific site (or node) in the system. The concept of preferred site is meant to favor transactions accessing objects maintained by the local nodes. With that, Walter can quickly commit these transactions without checking other nodes for write conflicts.

In other words, if a local transaction issues an operation on object x, then it can access the latest version of x. However, non-local transactions are still allowed to modify x on N_i but their updates can be repeatedly aborted in case the accessed version of x is not the latest one.

After a local transaction commits, the acknowledgment of its successful commit should be propagated to other nodes in the system. This propagation is done asynchronously and its goal is to eventually allow non-local transactions to advance their reading snapshot. As an example of the above propagation mechanism, suppose the preferred site of object x is N_1. Local transaction T_1 starts at N_1 and creates a new version x_1 of x. A non-local transaction T_2, started at node N_2, cannot create another version of x (i.e., x_2) until N_2 is being acknowledged about the commit of T_1 in N_1. After N_2 receives the propagation message of T_1's commit, T_2 is able to proceed its execution and successfully create x_2.

3.2 The Challenge of Updating Reading Snapshot in Walter

Walter does not update the reading snapshot of a transaction during its execution. This is because, by doing that without leveraging additional metadata, a well-known anomaly called Read Skew [3] might occur. Read Skew happens if a transaction T_1 reads a version x_1 for object x and concurrently a transaction T_2 commits an update on objects x and y, which creates a new version x_2 of x and y_2 of y. If T_1 reads y after committing T_2, by simply advancing its reading snapshot it might return y_2, which is incorrect.

Solutions in literature, such as SSS [18], GMU [24], and Nemo [21], overcome the Read Skew anomaly by updating vector clocks in a way that takes into account causal dependencies among nodes that have been previously contacted by a transaction. Walter prefers a simpler approach in which only the vector clock entry associated with the node where the transaction executes is updated upon commit. Such a decision is supported by the fact that read operations in Walter can read arbitrarily old values, therefore there is no need to account for causal dependency relations developed after the chosen reading snapshot. FW-KV's goal is to preserve the advantage of Walter's simpler concurrency control while adding additional metadata to improve data freshness of read-only transactions.

3.3 Data Freshness and the Long Fork Anomaly

Figure 1 shows an example of an execution accepted by Walter in which two read-only transactions are allowed to see the results of two update transactions in different order. Although this execution is admitted by PSI (anomaly known as long-fork [28]), it introduces an undesirable behavior at the application level, as described below.

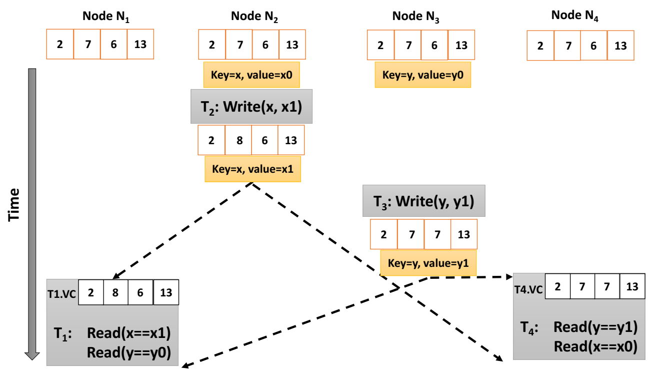


Figure 1: Dashed arrows represent the asynchronous propagation messages. The reading snapshot of T_1 reflects the timestamp of T_2 in the second entry of T_1's vector clock ($T_1.VC$) but it does not reflect the timestamp of T_3 in the third entry of T_1's vector clock. The reading snapshot of T_4 reflects the timestamp of T_3 in the third entry of T_4's vector clock ($T_4.VC$) but it does not reflect the timestamp of T_2 in the third entry of T_4's vector clock.

In the example we assume four nodes, N_1, N_2, N_3, N_4, and four transactions, T_1, T_2, T_3, T_4, each begins and executes on the respective node. By assumption, T_2 and T_3 are non-conflicting local update transactions; while T_1 and T_4 are non-local read-only transactions both accessing objects from N_2 and N_3. As of Walter's rule, each read-only transaction starts its execution by acquiring the latest vector clock of the node where it executes.

Both T_2 and T_3 after their commit on their preferred sites N_2 and N_3 send a propagation message to all other nodes. Let us assume T_1 starts its execution after receiving the propagation of T_2 and before receiving the propagation of T_3. On the other hand, T_4 starts its execution after receiving the propagation of T_3 and before receiving the propagation of T_2. Receiving propagate from different nodes in different orders is a likely scenario in an asynchronous distributed system.

Since T_1 and T_4 start after the commit of T_2 and T_3, their respective clients might have had the chance to interact with each other outside the system (e.g., in a social media platform when a user publishes a new post and alerts her/his friends about the new content so that they can read it). The consequence of this interaction is that T_1's and T_4's clients will not expect to observe a snapshot in which only some of the updates that they expected to be committed are returned by their read-only transactions.

FW-KV overcomes the above issue by allowing T_1 and T_4 to read the modifications made by T_2 and T_3, as long as *i)* no other reads accessing objects on N_2 and N_3 are issued by T_1 and T_4, and *ii)* T_2 and T_3 commit before T_1 and T_4 start. Note that, in the case T_1 and T_4 are concurrent with T_2 and T_3, both FW-KV and PSI allow T_1 and T_4 to observe update transactions in different order, therefore long fork is still possible for FW-KV as well. However, the latter case of long fork cannot trigger the behavior illustrated above at the client side, and this is again thanks to FW-KV's improve data freshness.

4 FW-KV: PROTOCOL DESCRIPTION

4.1 Metadata

Since FW-KV is built on top of Walter, we first list Walter's metadata for completeness and then we show the additional metadata required by FW-KV.

Transaction vector clock. A transaction T holds a vector clock `T.VC` whose size is equal to the number of nodes in the system. `T.VC` encapsulates the knowledge of T with respect to the logical timestamps of other nodes. In practice, `T.VC` is used as visibility bound for all versions accessible by T.

Transaction write-set. Every transaction T holds a private buffer called *T.writeset*, which contains the objects the transaction wrote, along with their values.

Current sequence number. Every node N_i is assigned with a number $CurrSeqNo_i$ representing the sequence number of the latest transaction issued and committed at node N_i.

The following metadata is exclusive for FW-KV.

Transaction node access vector clock. A transaction T records the sites where it reads from in a vector clock, called `T.hasRead`. Every time T reads from a node N_j for the first time during its execution, `T.hasRead[j]` is set to `true`. When `T.hasRead[j]` is set to `true`, T's visible timsestamp with respect to N_j is fixed and cannot be advanced for T's future accesses to N_j.

Node vector clock. Each node N_i is associated with a vector clock, called $siteVC_i$. The j^{th} entry of this vector clock represents the last transaction from node N_j that was committed at site N_i.

Transaction commit vector clock. When the commit decision for transaction T issued by N_i is made, the $CurrSeqNo_i$ is incremented and $siteVC$ of N_i is updated at the i^{th} position and the updated value of $siteVC$ is assigned to transaction commit vector clock (i.e., $T.commitVC$). In addition, the $CurrSeqNo_i$ is also sent to the other nodes involved in the commit procedure to update their $siteVC$ at the i^{th} position.

Version's vector clock. As it is mentioned in Section 2, each object o is associated with a set of versions where each version v is created by an update transaction. The commit vector clock of each update transaction is assigned to its created versions and is called version vector clock ($v.VC$).

Version identifier. Each version v of object o is associated with a monotonically increasing scalar number, called $v.id$.

Version access set. As shown in Section 3, by relying on the way Walter establishes transactions' commit vector clocks (i.e., without tracing causal dependencies among involved nodes), advancing transaction vector clock during execution without additional metadata violates PSI.

In order to advance the reading snapshot, given a transaction T_i the concurrency control needs to be able to trace concurrent transactions T_j that overwrite versions read by T_i. In this case we say that T_i has an anti-dependency relation (i.e., a read-after-write conflict) with T_j. FW-KV does that by implementing a technique called *visible reads* [22].

The visible reads technique is implemented in the following way. Each version is associated with a set containing identifiers of read-only transactions that read that specific version. During the commit phase of an update transaction, the set of identifiers of concurrent conflicting read-only transactions is collected. This set is propagated to the version-access-sets of the newly created versions of this update transaction since with its commit, it establishes transitive anti-dependency relations with those read-only transactions.

If a read-only transaction T contacts a node for the first time, it can advance its reading snapshot unless it finds that its own identifier exists in the version-access-set of the version to be read. In that case, T should select a previous version whose version-access-set does not contain T's identifier.

Algorithm 1 Begin procedure of transaction T in node N_i

```
1: function BeginTx(Transaction T)
2:     $T.VC \leftarrow siteVC_i$
3:     for all ($T.hasRead[i]$) do
4:         $T.hasRead[i] \leftarrow false$
5:     end for
6: end function
```

4.2 Transactional Begin and Write Operation

Alg. 1 represents the way that transaction T vector clocks ($T.hasRead$ and $T.VC$) are initialized once T begins. When T begins in node N_i, it assigns the $siteVC$ of N_i, which shows the vector clock of the latest committed/propagated transactions from all the sites in/to N_i, to its own $T.VC$. At this point, since no read is issued yet, all elements of $T.hasRead[j]$ are set to `false`.

In FW-KV update transactions implement lazy update, meaning their written keys are not immediately visible and accessible at the time of the write operation, but they are buffered in the transaction *writeset*.

4.3 Transactional Read Operation

Alg. 2 describes the steps of a read operation for key k by transaction T. If k has been already written by transaction T, then the written value of k is returned (Lines 2- 4 of Alg. 2). Otherwise, a read request (`ReadRequest`) is forwarded to the node that stores k, which might be the same node where T executes (local read) (Line 6 of Alg. 2).

The read is handled differently depending on the type of the issuing transaction. Importantly, for avoiding concurrent modifications while the read logic is processed, the read handler should be executed in mutual exclusion with respect to message handlers from other concurrent conflicting update transactions. However, read-only transactions are still allowed to operate simultaneously on read handlers.

Algorithm 2 Read operation

```
 1: function READ($Transaction\ T, key\ k$)
 2:     if $< k, val > \in T.writeset$ then
 3:         return $val$
 4:     end if
 5:     $target \leftarrow site(k)$
 6:     send $ReadRequest[T, k]$ to $target$
 7:     wait Receive $ReadReturn[val, maxVC]$ from $target$
 8:     $T.hasRead[target] \leftarrow true$
 9:     $T.VC \leftarrow max(T.VC, maxVC)$
10:     if (T is read-only) then
11:         $T.readKeys \leftarrow T.readKeys \cup \{k\}$
12:     end if
13:     return $val$
14: end function
```

Read operations by Read-only Transactions. Lines 2-10 of Alg. 3 describes the read policy for a read-only transaction T. The first step is to identify the set of versions for k that are visible according to $T.VC$. We say a version v is *visible* for a transaction T if all the entries of $T.VC$, for which $T.hasRead$ is `true`, have values greater or equal to the values of the respective entries in $v.VC$ (Alg. 3 Lines 4).

Algorithm 3 Version selection logic in node N_i

```
 1: upon Receive $ReadRequest[T, k]$ from $N_j$ in $N_i$ do
 2:     if ($T$ is read-only) then
 3:         get lock($key = k, owner = T.id$)
 4:         $VisibleSet \leftarrow \{v \in k.versionSet : \forall site \in sites :$
              $T.hasRead[site] = true \Rightarrow v.VC[site] <=$
              $T.VC[site]\}$
 5:         $ExcludedSet \leftarrow \{v \in VisibleSet : T.id \in v.accessSet\}$
 6:         $VisibleSet \leftarrow VisibleSet \backslash ExcludedSet$
 7:         $version \leftarrow ver \in VisibleSet : \forall v \in VisibleSet \Rightarrow$
              $ver.id >= v.id$
 8:         $version.accessSet \leftarrow version.accessSet \cup \{T.id\}$
 9:         release lock($key = k, owner = T.id$)
10:     end if
11:     if ($T$ is update) then
12:         get lock($key = k, owner = T.id$)
13:         $VisibleSet \leftarrow \{v \in k.versionSet : \forall site \in sites :$
              $T.hasRead[site] = true \Rightarrow v.VC[site] <=$
              $T.VC[site]\}$
14:         $ExcludedSet \leftarrow \{v \in VisibleSet : \forall site \in sites :$
              $T.hasRead[site] = true \Rightarrow v.VC[site] = T.VC[site]$
              $\wedge \exists s \in sites : T.hasRead[s] = false. \wedge v.VC[s] > T.VC[s]\}$
15:         $VisibleSet \leftarrow VisibleSet \backslash ExcludedSet$
16:         $version \leftarrow ver \in VisibleSet : \forall v \in VisibleSet \Rightarrow$
              $ver.id >= v.id$
17:         release lock($key = k, owner = T.id$)
18:     end if
19:     send $ReadReturn\ [version, version.VC]$ to $N_j$
20: end
```

From the latter set ($VisibleSet$ in the Alg. 3), those versions whose version-access-set include T's identifier, should be excluded because that means T has already established an anti-dependency (directly or transitively) with the transactions that committed those versions. Among the remaining versions, the one with the highest identifier (meaning the freshest among them) is selected as the result of the read operation.

Figure 2 illustrates an example of how read-only transactions establish their reading snapshots. Transaction T_1 starts its execution at node N_1 and reads x_0, the latest version of object x, when it accesses node N_2 (Lines 2-10 of Alg. 3). Note that the *ExcludedSet* in Line 5 of Alg. 3 is empty. Upon reading x_0, the identifier of T_1 is inserted into the corresponding version-access-set of x_0 (Line 8 of Alg. 3). T_1 also updates $T_1.VC[2]$ to the latest timestamp of N_2 which is "7" (Line 9 of Alg. 2). After that, a concurrent update transaction T_3 commits an update on x and y on N_2 and increments $siteVC_2[3]$ to timestamp "7" (Line 21 of Alg. 5). Later, after T_3 commits at N_2, T_1 issues another read on y. At this point, since y_1's version-access-set includes T_1's identifier, because it has been inserted by the commit procedure of T_3 (done in Line 19 of Alg. 5 – see Section 4.4), y_1 cannot be returned by T_1's read operation due to the anti-dependency relation already established between T_1 and T_3 (*ExcludedSet* includes T_3's identifier in Line 5 of Alg. 3).

After committing, a `Remove` message to N_2 is sent for notifying the completion of T_1 (see Section 4.5). A read-only transaction should also record the accessed keys in a set called `readkeys`, used only to dispatch `Remove` messages.

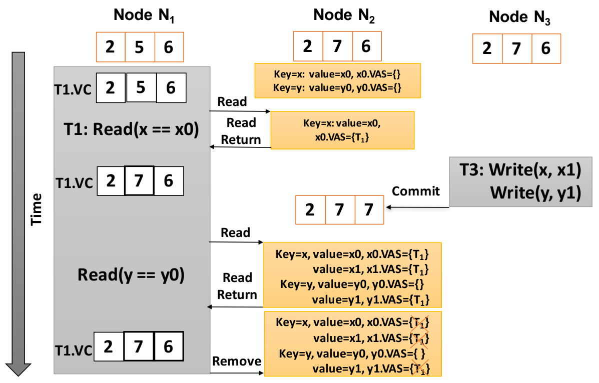

Node N_1
Node N_2
Node N_3
2 5 6
2 7 6
2 7 6
Key=x: value=x0, x0.VAS={}
Key=y: value=y0, y0.VAS={}
T1.VC 2 5 6
Read
T1: Read(x == x0)
Read Return
Key=x: value=x0, x0.VAS={T_1}
T1.VC 2 7 6
T3: Write(x, x1) Write(y, y1)
Commit
2 7 7
Time
Read
Read(y == y0)
Read Return
Key=x, value=x0, x0.VAS={T_1}
value=x1, x1.VAS={T_1}
Key=y, value=y0, y0.VAS={}
value=y1, y1.VAS={T_1}
T1.VC 2 7 6
Remove
Key=x, value=x0, x0.VAS={$\cancel{T_1}$}
value=x1, x1.VAS={$\cancel{T_1}$}
Key=y, value=y0, y0.VAS={ }
value=y1, y1.VAS={$\cancel{T_1}$}

Figure 2: Example of execution where a read-only transaction advances its reading snapshot and still reads consistently. VAS is the version-access-set. Bold vector clock entries show where *hasRead* is `true`. The red crossed entries of VAS represent their elimination upon Remove.

Read operations by Update Transactions. Update transactions do not insert their identifier in the version-access-set of their read keys. However, upon their first read operation, they still advance their reading snapshot to be able to observe the accessed object. Subsequent read operations will use the same reading snapshot without updating it.

Lines 11-18 of Alg. 3 show the pseudo code for handling read operations by an update transaction T. The $VisibleSet$ is determined as follows. First, the versions that are visible according to

$T.VC$ are selected. From them, the versions produced by concurrent transactions with anti-dependency with T should be excluded. However, since the version-access-set cannot be leveraged to precisely identify anti-dependency relations, as the case of read-only transactions, we adopt a more conservative condition for version exclusion, inspired by [23], which over-approximates the existence of an anti-dependency by just comparing T's vector clock against the candidate version's commit vector clock.

A version should be excluded if it has a vector clock in which, in all the positions where $T.hasRead$ is `true`, the value is equal to the value of the same entry in $T.VC$ (Lines 13-15 of Alg. 3) and there exist at least one position in $T.VC$ whose corresponding entry in $T.hasRead$ is `false` and in the same position the version vector clock has a greater value than $T.VC$. The latter clause of the above condition allows an update transaction to also exclude a version committed by a concurrent transaction, or a transaction whose acknowledgment has not been received yet, without an anti-dependency with T, which is a false positive case since that version could be read without compromising PSI.

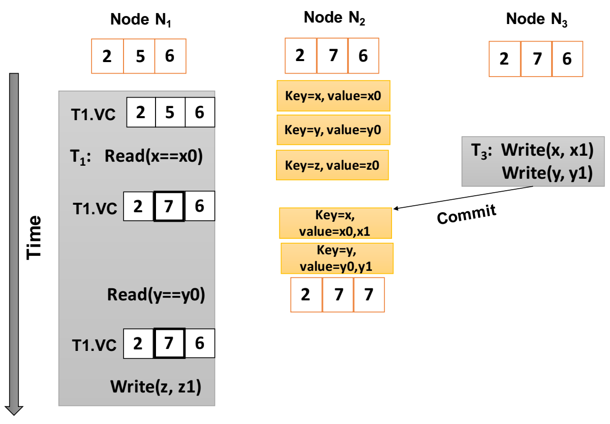


Figure 3: Example showing how an update transaction establishes its reading snapshot.

After that, the version with the highest identifier in the resulting $VisibleSet$ is returned as the result of the read operation (Line 16 of Alg. 3), along with its vector clock.

When the response for T's read operation is returned to N_i, an entry-wise maximum between $T.VC$ and the version vector clock is performed to advance the reading snapshot of T (Line 9 of Alg. 2).

Figure 3 shows an example of how update transactions establish their reading snapshot. We have two update transactions T_1 and T_3. T_1 reads x_0, the latest version of object x at its first access to node N_2 (Line 11- 18 of Alg. 3). At this stage, the *ExcludedSet* in Line 14 of Alg. 3 is empty. T_1 then advances its reading snapshot by updating the second entry of $T_1.VC$ to "7" (Line 9 of Alg. 2). Concurrently T_3 updates both objects x and y, stored on N_2, and commits by advancing N_2's vector clock at its third entry to "7" (Line 21 of Alg. 5).

After that, T_1 performs its second read operation on y. Here, T_1 cannot read version y_1. This is because $T_1.VC[2]$ is equal to $y1.VC[2]$ and $T_1.hasRead[2]$ is `true`. In this case, since $T_1.VC[3]$ is less than $y1.VC[3]$, it might mean that y_1 has been committed by a concurrent conflicting transaction. In fact the *ExcludedSet* disallows T_1's second read operation from accessing version y_1 because y_1 includes timestamp "7" at $y1.VC[3]$ (Line 14 of Alg. 3). However, due to the way vector clocks are incremented upon commit, T_1 does not have enough knowledge to verify if y_1's committer was a conflicting transaction. Therefore the read operation returns a safe snapshot for T_1, which in this case is y_0 because y_0's vector clock (i.e., $y_0.VC$) is visible by T_1. Note that in this case even if T_3 only updates y (which means no conflict between T_1 and T_3), T_1 still cannot return y_1.

4.4 Commit protocol

The commit phase of transaction T is performed through the `COMMIT` function in Alg. 4. If T is a read-only transaction, the commit phase only consists of a clean up step to remove traces of its execution on the version-access-set of its read versions. To do that, `Remove` messages are sent to the nodes where T read from (Lines 2-8 of Alg. 4).

Algorithm 4 Commit of transaction T in node N_i

```
1:  function COMMIT(Transaction T)
    // Check if T is a read-only transaction
2:      if (T.writeset=$\phi$) then
3:          for ($k \in T.readKeys$) do
4:              Send Remove[$T.id, k$] site(k)
5:          end for
6:          $T.outcome \leftarrow true$
7:          return $T.outcome$
8:      end if
    // Start 2PC if T is an update transaction
9:      $commitVC \leftarrow T.VC$
10:     $T.collectedSet \leftarrow \phi$
11:     $T.outcome \leftarrow true$
12:     send Prepare[$T$] to all $N_j \in sites(T.writeset)$
13:     for all ($N_j \in sites(T.writeset)$) do
14:         wait receive Vote[$collectedSet_j, result_j$] from $N_j$ or timeout
    // Check if T's 2PC commit decision is successful
15:         if ($\neg result_j \vee timeout$) then
16:             $T.outcome \leftarrow false$
17:             break;
18:         else
    // Collect all existing anti-dependencies in $T.collectedSet$
19:             $T.collectedSet \leftarrow T.collectedSet \cup collectedSet_j$
20:         end if
21:     end for
22:     $currSeqNo_i \leftarrow currSeqNo_i + 1$
23:     $T.seqNo \leftarrow currSeqNo_i$
    // Finalize T's commit vector clock
24:     $T.commitVC \leftarrow siteVC_i$
25:     $T.commitVC[i] \leftarrow T.seqNo$
26:     send Decide[$T, T.outcome$] to all $N_j \in sites(T.writeset \cup N_i)$
27:     send Propagate[$T, T.seqNo$] asynchronously to all
        $N_j \in sites \backslash sites(T.write)$
28:     return $T.outcome$
29: end function
```

If T is an update transaction, similar to Walter the Two-Phase Commit (2PC) protocol is used to accomplish the commit phase and install new versions into the data repository. The node in which T executes (i.e., T's coordinator) starts the 2PC by sending a `Prepare` message to the (preferred) nodes that store the objects written by T (Line 12 of Alg. 4). When a 2PC participant node N_i receives a `Prepare` message for T, all the written objects by T and stored by N_i are locked. If the locking acquisition succeeds, then versions are validated to certify that they have not being overwritten meanwhile.

At this point, the existing read-only transactions' identifiers in the versions-access-set of T's written objects are retrieved by the

2PC participants and sent back to the 2PC coordinator with the `Vote` message (Lines 3-12 of Alg. 5). Once the coordinator receives all the `Vote` messages from participants, it merges all the received transactions' identifiers and include them into $T.collectedSet$ (Line 19 of Alg. 4).

In the case all participants vote for committing T, meaning they were able to acquire locks on the written objects and validate their version, then N_i's sequence number ($CurrSeqNo_i$) is incremented and the commit vector clock of T is established. This vector clock is then sent along with the `Decide` message to the 2PC participants (Line 22-26 of Alg. 4).

Algorithm 5 Commit message handlers received by node N_i for transaction T issued by node N_j

```
1:  upon receive Prepare[$Transaction\ T$] from $N_j$ do
2:      $collectedSet \leftarrow \phi$
        // Check if T passes lock acquisition and validation
3:      boolean $result \leftarrow getLocks(T.writeset, owner = T.id)$
        $\wedge validate(T)$
4:      if ($\neg result$) then
5:          $releaseLocks(T.writeset, owner = T.id)$
6:          send Vote[$collectedSet, result$] to $N_j$
7:      else
8:          for all $k \in T.writese$ do
9:              $collectedSet \leftarrow collectedSet \cup k.version.accessSet$
10:         end for
11:         send Vote[$collectedSet, result$] to $N_j$
12:     end if
13: end
14: upon receive Decide[$T, outcome$] from $N_j$ do
15:     if ($outcome$) then
16:         wait until $siteVC_i[j] = T.seqNo - 1$
17:         $update(T.writeset,\ T.seqNo,\ j)$
18:         for all ($k \in T.writeset$) do
19:             $k.lastVersion.accessSet \leftarrow k.lastVersion.accessSet \cup$
                 $T.collectedSet$
20:         end for
21:         $siteVC_i[j] \leftarrow T.seqNo$
22:         $releaseLocks(T.writeset, owner = T.id)$
23:     else
24:         $releaseLocks(T.writeset, owner = T.id)$
25:     end if
26: end
27: function validate($Transaction\ T$)
28:     for all ($k \in T.writeSet$) do
29:         if ($k.lastVersion.VC[lastUpdaterSite] >$
             $T.VC[lastUpdaterSite]$) then
30:             return false
31:         end if
32:     end for
33:     return true
34: end function
```

Lines 14-26 of Alg. 5 show the steps taken by a 2PC participant N_i when it receives the `Decide` message from the coordinator executing on node N_j. In order for N_i to commit T, N_i must wait for all already decided/propagated transactions by N_j. N_i can easily detect if this wait condition should occur by looking at the gap between the node vector clock in position j and the commit vector clock of T at position j (e.g., $T.seqNo$). When T finally commits, the $siteVC$ of each 2PC participant is updated in the j^{th} position.

Similar to Walter, after sending the `Decide` of T FW-KV sends the asynchronous `Propagate` message to all other nodes in the system in order to allow them to advance their reading snapshot with respect to N_i. Note that, although FW-KV requires `Propagate` messages to commit non-local update transactions, it does not abort these transactions as Walter does due to late delivery of `Propagate` messages. In fact, in Walter if a `Propagate` message from a node N_j is not delivered by a node N_i, a non-local update transaction from N_i will repeatedly fail its validation step causing an abort that will be solved only after receiving the `Propagate` message.

FW-KV does not abort the update transaction in such a case. However, although it still needs the `Propagate` message to be delivered in order to finalize the commit, *i)* it is able to overlap the transaction execution with the delivery of the `Propagate` message, which is likely to arrive meanwhile; and *ii)* it reduces network traffic due to saving multiple transaction retries.

Figure 4 pictures an example of an update transaction that can commit in FM-KW but that would be aborted by Walter due to reading outdated versions. Transaction T_1 is an update transaction that starts its execution on node N_1. Upon contacting node N_2 for reading object x, T_1 is able to access x_1, the latest version of x. In order to commit, T_1 performs the steps shown in Lines 1- 12 of Alg. 5. It is able to successfully pass the validation procedure in Lines 27- 34 because of the value of $T_1.VC$, which was updated upon reading x_1 (Line 9 of Alg. 2). Without that update, T_1 would need to abort, as the case of Walter.

4.5 Handling Asynchronous Messages

Alg. 6 shows how node N_i handles asynchronous messages, namely `Propagate` and `Remove`. When N_i receives a `Remove` message because a read-only transaction T committed at node N_j, T's identifier is removed from the version-access-sets of T's read versions whose preferred site is N_j, and from all other version-access-sets in N_j in which T's identifier has been propagated by concurrent update transactions that committed meanwhile (Lines 5-10 of Alg. 6).

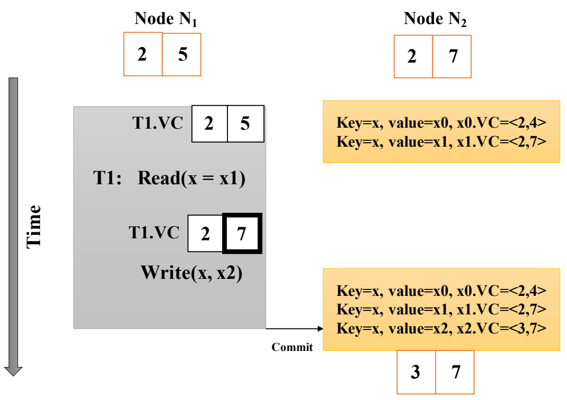

Node N1
2 5
Node N2
2 7
Time
T1.VC 2 5
T1: Read(x = x1)
T1.VC 2 7
Write(x, x2)
Key=x, value=x0, x0.VC=<2,4>
Key=x, value=x1, x1.VC=<2,7>
Key=x, value=x0, x0.VC=<2,4>
Key=x, value=x1, x1.VC=<2,7>
Key=x, value=x2, x2.VC=<3,7>
Commit
3 7

Figure 4: Example of saving aborts due to reading fresher snapshots in update transactions. For the sake of simplicity, version-access-sets have been omitted. Update of T_1's vector clock is shown in bold square.

Upon receiving a `Propagate` message by node N_i for the commit of an update transaction T from node N_j, N_i can advance its reading snapshot with respect to N_j to $T.seqNo$.

In PSI the outcome of all committed transactions that update some objects whose preferred site is N_j should be observed in the same order by N_i. For this reason, T should wait for all previously committed transactions in N_j with a lesser sequence number than $T.seq$ to be received by N_i (Line 2 of Alg. 6). After that, $siteVC$ of N_i

Algorithm 6 Remove & Propagate messages from transaction T issued by N_j to node N_i

```
1: upon receive Propagate[$T, T.seqNo$] from $N_j$ do
2:     wait until $siteVC_i[j] = T.seqNo - 1$
3:     $siteVC_i[j] \leftarrow T.seqNo$
4: end
5: upon receive Remove[$T.id,\ k$] from $N_j$ do
6:     $k.version.accessSet \leftarrow k.version.accessSet \setminus \{T.id\}$
7:     for all ($k' : v \in k'.versionSet \wedge T.id \in v.accessSet$) do
8:         $v.accessSet \leftarrow v.accessSet \setminus \{T.id\}$
9:     end for
10: end
```

can be updated with $T.seqNo$ at the j^{th} position of $siteVC$ (Line 3 of Alg. 6).

4.6 Correctness Arguments

We assess the correctness of FW-KV by discussing how our modifications on top of the Walter distributed concurrency control still preserve PSI.

The major difference between FW-KV and Walter lies on the fact that in FW-KV every transaction can read the latest version of its first accessed object, even if an asynchronous propagate message has not being delivered yet. Our approach is to focus on the necessary and sufficient condition to assess if an execution satisfies the Generalized Snapshot Isolation (GSI) correctness level [6]. GSI generalizes SI by allowing reading snapshots to be arbitrarily old, but still disallows PSI's long fork anomaly. Showing the equivalence to GSI is enough since we have already shown that FW-KV does not eliminate the long fork anomaly of Walter, as discussed in Section 3.3. Without considering this anomaly, PSI is equivalent to GSI [6, 28].

For a schedule to be accepted by GSI, if a transaction history has a cycle, then this cycle includes at least two adjacent anti-dependency edges in the Directed Serialization Graph [6].

Our correctness discussion shows that as soon as a transaction detects an anti-dependency with respect to a concurrent update transaction, a direct dependency, including a transitive one, cannot occur. This can be achieved by relying on either the content of the version-access-set (populated through the visible reads technique) for read-only transactions (Lines 4- 8 of Alg. 3), or the selection of a safe snapshot for update transactions (Lines 13- 16 of Alg. 3). As a consequence of this observation, only transactions executions with two adjacent anti-dependency edges can be committed by FW-KV, which is needed to satisfy GSI (and PSI by including the long fork anomaly). In fact, concurrent update transactions that are candidate to establish a direct dependency are excluded. In our algorithms, Line 5 of Alg. 3 refers to the case in which a read-only transaction excludes a concurrent update; while Line 14 of Alg. 3 refers to the case where an update transaction exclude a concurrent update.

Regarding the reading policy of read-only transactions, since a transaction T_{RO} that reads a version o_v of object o is included in the version-access-set (Line 8 of Alg. 3) of o_v, when an update transaction creates a new version o_{v+1}, the write-after-read (anti-) dependency is established and can be detected by any other reading transaction after that. That means, if a conflicting transaction, directly or transitively, produces a new version, that version cannot be returned by any subsequent read operation from T_{RO} because of the way the version-access-set is propagated to conflicting transactions, including those transitive (see Lines 18-20 of Alg. 5). By doing that, there cannot be a read-only transaction involved in a loop with an outgoing anti-dependency edge preceded by an incoming direct dependency edge. In the presence of an established anti-dependency, our concurrency control reads previous versions, which transforms the above direct dependency into an anti-dependency, as demanded by PSI.

The argument for an update transaction T is simpler since it cannot always attempt to access the latest version of an object. In fact, after the first read operation, a safe reading snapshot is established for T. This safe snapshot is established as follows. After the first read operation served by node n, for any subsequent operation requiring access to a node s, with $n \neq s$, the following check in Line 14 of Alg. 3 (i.e., $\exists s \in sites : T.hasRead[s] = false. \wedge v.VC[s] > T.VC[s]\}$) excludes the versions v for which $VC[s]$ has a value greater than $T.VC[s]$. Such a reading snapshot guarantees that if a concurrent transaction T' overwrites a read version by T, since T's vector clock will be strictly lesser than T''s vector clock, T cannot include that newer version in its reading snapshot. (Recall that this conservative rule might produce false conflicts that can unnecessarily order T before T' as mentioned is Section 4.3).

5 EVALUATION STUDY

FW-KV's distributed concurrency control has been embedded into an in-memory distributed transactional key-value store. We use the code base of Walter available at [18] and we modify it to integrate FW-KV's metadata and reading/writing policy. We recall that our performance assessment for FW-KV aims at showing how its algorithmic modifications, which ensure higher level of freshness than Walter, can still provide comparable performance with respect to Walter, and retain significant performance improvement over a serializable distributed concurrency control [16].

We conduct the performance evaluation using two well-known OLTP benchmarks, YCSB [8] and TPC-C [10], both ported to the key-value data model. For YCSB, we have two transaction profiles: update, where two keys are read and written, and read-only transactions, where two keys are accessed. YCSB is configured to use keys of 4 bytes and values of 12 bytes. TPC-C is a more complex benchmark that simulates an order-entry environment with several warehouses. It includes five transaction profiles, three of them are update transactions and the remaining are read-only transactions.

We configure the benchmarks to explore different runtime scenarios. First, YCSB transactions are shorter than TPC-C's transactions; also, since update transactions in YCSB write the same keys they read, the final execution is equivalent to an execution in which the concurrency control ensures Serializability. This is done to particularly stress the importance of reading a fresh snapshot for update transactions. In fact, FW-KV will be able to reduce the number of aborts of update transactions due to outdated reading snapshot in Walter. On the other hand, TPC-C transactions' logic allows for reading and writing different shared objects, showing a favorable case for Walter since an outdated reading snapshot still suffices to commit while preserving PSI.

We compare the performance of FW-KV against Walter, which guarantees PSI, and 2PC-baseline (2PC in the plots), a serializable key-value store where all transactions execute optimistically and rely on the Two-Phase Commit protocol to commit both update and read-only transactions, thus without needing multiversioning. We also included a version of Walter and FW-KV in which the asynchronous propagate messages are intentionally delayed to show the effect of such an event on the abort rate of update transactions.

In all the experiments there are five application threads (i.e., clients) per node injecting transactions in a closed-loop (i.e., a client issues a new request only when the previous one has returned). In terms of transaction mix, we evaluate our competitors using 20% and 50% read-only transactions. We do not include the test with 80% read-only transactions because performance of both Walter and FW-KV are almost identical using this configuration, especially when the contention is low. This is expected since most of the algorithmic differences between the two competitors are related to the propagation of anti-dependency developed with update transactions. If version-access-sets are almost empty, the performance of read-only transactions in both competitors will be similar.

In both benchmarks, transactions select keys to be accessed using a uniform distribution, which entails accesses might or might not be to the local data repository. We do not test the case of a skewed access distribution to highlight the performance impact of FW-KV design. In fact, if accesses target local nodes, data freshness is already guaranteed to be the highest level. In this scenario, FW-KV performs equally to Walter since no protocol modification has been made to Walter to improve freshness of local accesses. In terms of data distribution, keys are evenly distributed across nodes.

As test-bed, we use CloudLab [27], a cloud infrastructure available to researchers. We selected 20 nodes of type c6320 available in the Clemson cluster. This type is a physical machine with 28 Intel Haswell CPU-cores and 256GB of RAM. Nodes are interconnected using a 10Gb/s network, which delivers a message in about 20 microseconds without saturation. Considering that, we set the timeout on lock acquisition to 1 ms. All the results are the average of 5 trials.

5.1 YCSB

Figure 5 shows throughput (k transactions committed per second) of all competitors using YCSB and a total of 50k and 500k shared keys while increasing the total number of nodes. Recall that more nodes means more clients injecting transactions in the system, therefore an increasing level of contention. In all these configurations, the measured abort rate is below 10% at the highest contention level (i.e., 50k keys and 20 nodes).

The performance and scalability of FW-KV match Walter's in the cases where contention is low, namely up to 10 nodes in all tested cases and in the 500k configuration. When contention increases (e.g., due to higher number of clients), the gap between FW-KV and Walter becomes more visible. This is because of two factors: the additional synchronization steps needed by FW-KV's read operations, and the increasing size of version-access-sets (see Figure 6). Quantifying, for 20% read-only workload the highest gap measured between FW-KV and Walter is 20% and 16% with 50k and 500k keys, respectively. At 50% read-only workload, the gap is 15% at 50k keys, and such a gap is annulled at 500k keys.

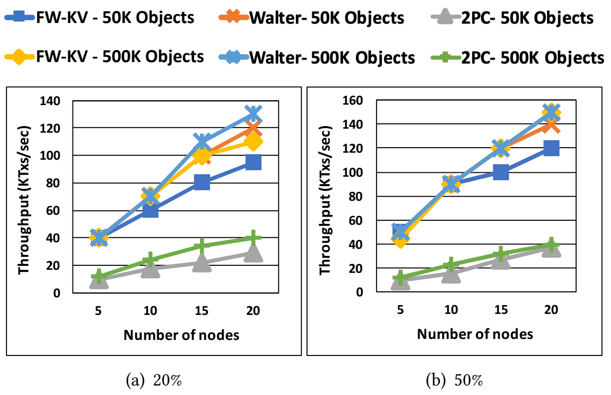


Figure 5: Throughput using YCSB and by varying % of read-only transactions, total keys, and number of nodes.

PSI competitors substantially improve performance over 2PC-baseline because its read-only transactions undergo an expensive commit phase using the 2PC protocol, which is skipped by FW-KV and Walter since their read-only transactions are abort-free. Achieved speedup of PSI competitors against 2PC-baseline is constantly more than 3x.

As observed earlier, the size of version-access-set impacts the gap in performance between FW-KV and Walter when the contention increases. Figure 6 confirms that. In this figure we report the average number of collected anti-dependency while an update transaction in FW-KV undergoes the prepare step of its commit phase. We explored the configurations with 20%, 50%, and 80% of read-only transactions, with 50k, 100k, and 500k shared objects.

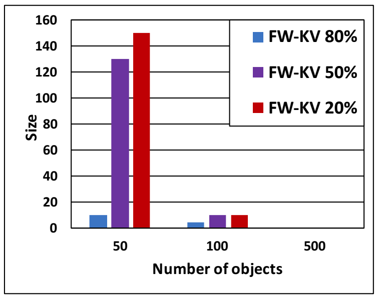


Figure 6: Average size of anti-dependency collected by update transactions in FW-KV during prepare phase for different % of read-only transactions and keys.

Increasing the percentage of update transactions increases the number of anti-dependencies. The sharp jump from 80% to 50% read-only at 50k keys is due to the transitive propagation of those anti-dependencies. In fact, if an update transaction reads a key whose version-access-set includes a number of read-only transaction identifiers, this set will be propagated to the version-access-set of the new written versions of the update transaction upon its commit.

In Figure 6, we also test the cases of 100k and 500k keys to show how the size of collected anti-dependencies gradually decreases to zero, as with 500k. YCSB transactions are short, therefore the chance for an anti-dependency to occur at the low contention case, such as using 500k keys, is low. Figure 6 also helps assessing the space overhead of FW-KV compared to Walter. The dominant factor in this case is recording the transactions' version-access-sets. It is clear from the figure that unless contention is high (e.g., 50k objects) and the workload is dominated by update transactions, the version-access-sets are likely empty or storing very few items. As a result, in these configurations FW-KV and Walter have comparable storage cost.

Another conclusion that can be drawn by analyzing the results of Figure 6 is that transaction latency for read-only workload is comparable with the one in Walter. In fact, querying and manipulating version-access-sets might add latency to read-only transactions. However, under read-only heavy workload and low contention, version-access-sets are effectively empty, which minimizes FW-KV's latency overhead with respect to Walter.

To show the effectiveness of a fresher reading snapshot for read operations of update transactions, in Figure 7 we measure the abort rate (of update transactions since read-only transactions cannot abort) using 20 nodes in case we intentionally delay the asynchronous propagate messages (by 1 ms) in both FW-KV and Walter. We select 1 ms because, in our testbed it mimics around 5x slowdown of network delay, which might be due to congestion at high utilization.

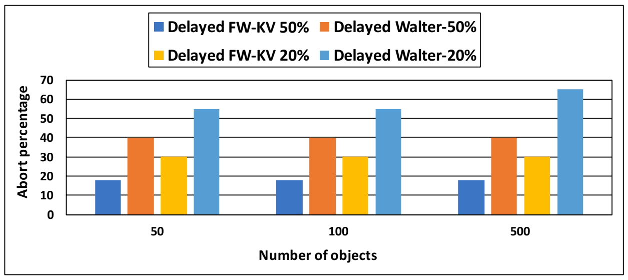


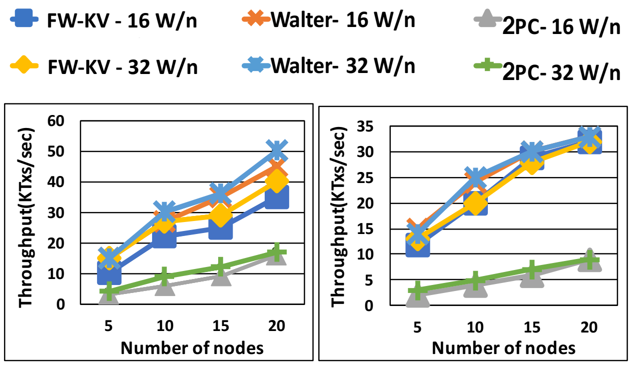
Figure 7: Abort rate using 20 nodes and varying number of keys while delaying propagate messages.

Without delaying the asynchronous propagate messages, the abort rate of FW-KV and Walter is comparable, below 10% and even less in low-contention scenario. Enabling the delay, Walter's abort rate is on average twice the one of FW-KV. The reason of such significant increase for Walter is because update transactions' reading snapshot in our configuration of YCSB should be the freshest since the same read keys are also written, therefore they need to be validated. Slowing down the propagate messages forces update transactions in Walter to repeatedly abort before being able to commit when finally the node's vector clock is updated. Another interesting aspect to be observed is that in general the abort rate does not decrease while the contention decreases at 500k keys. This is due to the fact that, even if contention is absent, a transaction in Walter may not be able to read the latest version of a key because of an outdated node vector clock.

Abort rate increases in Walter and FW-KV compared to the case where asynchronous messages are not delayed because update transactions still need to receive the propagate messages in order to finally commit. While they wait for such a message, they hold the locks on their written keys. Holding locks for longer increases the probability of abort.

5.2 TPC-C

TPC-C transactions are much longer than YCSB's, especially the read-only ones. Generally, the performance at 50% read-only workload is slower than the one at 20%. Because of the hierarchical object access pattern of TPC-C, the contention in the system is modified by varying the number of warehouses (the warehouse object sits at the top of this access hierarchy).

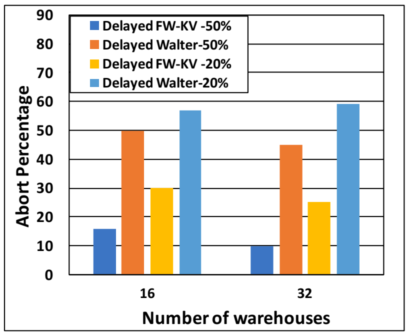


(a) 20% (b) 50%

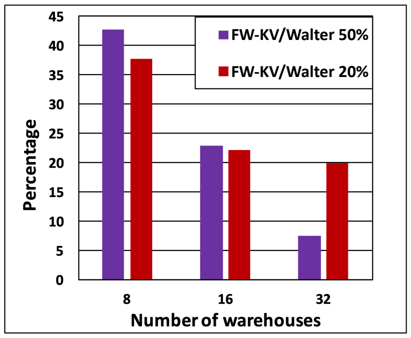
Figure 8: Throughput using TPC-C and by varying % of read-only transactions, the number of warehouses per node (W/n), and the number of nodes.

Figure 8 shows the results for all competitors varying the number of nodes and the number of warehouses per node. As opposed to YCSB benchmark, in TPC-C transactions do not necessarily read the same keys that they write. This allows an update transaction to commit even if the reading snapshot is not the freshest. The consequence of this characteristic is that PSI competitors are much faster than 2PC-baseline, and both Walter and FW-KV have similar growing trend. In fact, with 50% read-only transactions, the performance of the two PSI competitors is within 5% of each other. At 20% read-only workload, the maximum observed gap is 28%.

Figure 9(a) includes the abort rate measured at 20 nodes deploying 16 and 32 warehouses per node in the case where the propagate messages have been intentionally delayed. Without delaying them, abort rate of Walter and FW-KV is comparable. Walter shows an average of almost 4x higher abort rate than FW-KV. This is because of the way the safe snapshot is selected by update transactions in FW-KV. In fact, according to TPC-C logic, the warehouse is often the first accessed key, which is guaranteed to be the latest version by FW-KV's concurrency control, subsequent accesses to objects will be likely related to that warehouse. This pattern ensures that all the objects updated along with that warehouse will be accessed by reading the latest version. Because of that, FW-KV's degradation in abort rate is less than Walter's. In terms of throughput (not shown in the plots), results of the delayed version of both FW-KV and Walter are consistent with the trends observed in Figure 8.



(a) Abort rate



(b) Slowdown

Figure 9: Performance of FW-KV and Walter varying the number of warehouses per node.

Finally, in Figure 9(b) we show the slowdown in throughput between FW-KV and Walter when we vary the number of warehouses, using 20 nodes. In TPC-C, every read-only transaction needs to be added to the read-access-set of the accessed warehouse. As a result, when the number of warehouses is only 8 per node, contention is high and the size of read-access-sets increases along with the number of read-only transactions. Managing large read-access-sets introduces overhead, which is why with 8 warehouses, performance at 20% read-only workload is slightly better than at 50%. The trend reverses as the number of warehouses increases and contention decreases.

6 RELATED WORK

Many distributed transactional repositories providing either SI or its weaker variants have been proposed in literature; examples include [1, 5, 12, 13, 25, 28]. Among those, Jessy [1], Clock-SI [12], Percolator [25], and the Incremental approach [5] will be discussed.

Jessy [1] provides transactions with reading snapshots that can include causally dependent versions committed after the transaction starting time. Jessy uses per-version dependence vectors. Each vector reflects all the versions read or written by the transaction that created that specific version. FW-KV and Jessy both aim at improving data freshness; however, unlike FW-KV, the amount of metadata required to support execution can grow significantly. In fact, if transactions access random objects, the size of each dependence vector becomes comparable to the number of objects in the system.

Clock-SI [12] provides SI using a loosely synchronized clock scheme that might lead to unavailability of the reading snapshot because of skews across distributed clocks, with a consequence low performance. Google Percolator [25] provides SI using a centralized source of synchronization to timestamp distributed transactions for Bigtable [7].

Elnikety et. al in [13] extend SI to replicated databases. It allows transactions to use local snapshots of the database on each replica and relaxes the level of data freshness.

The solution in [5] proposed the Incremental Snapshot method as an efficient solution to implement Distributed Snapshot Isolation. In this method, a local transaction only interacts with the local clock to establish the reading snapshot. A non-local transaction interacts with the remote node to obtain an appropriate reading snapshot. For validating the remote accesses, a global clock is still required. The validation requires maintaining the mapping between each local clock and the global clock.

Among the proposed distributed systems relying on special purpose hardware for synchronization or communication, Spanner [9], Farm [11] and FaSST [17] can be considered state-of-the-art. Spanner [9] relies on TrueTime API which uses a combination of a very fast dedicated network, GPS, and atomic clocks to provide a fresh reading snapshot. FaRM [11] and FaSST [17] are distributed computing platforms which use RDMA to directly access data in a shared address space, and for fast messaging between the nodes. FW-KV is designed to not leverage special purpose hardware.

7 CONCLUSION

We presented FW-KV, a distributed concurrency control that improves upon Walter's PSI concurrency control by increasing the level of data freshness of read-only transactions. With FW-KV, we empirically show that is possible to retain the high performance enabled by PSI while preventing transactions from reading arbitrarily old versions, a significant drawback of current state-of-the-art PSI solutions.

8 ACKNOWLEDGMENTS

Authors would like to thank our shepherd Roy Friedman and the anonymous reviewers for their insightful comments. We are also grateful for the feedback received by dePaul Miller. This material is based upon work supported by the Air Force Office of Scientific Research under award number FA9550-17-1-0367 and by the National Science Foundation under Grant No. CNS-1814974.

REFERENCES

[1] Masoud Saeida Ardekani, Pierre Sutra, and Marc Shapiro. 2013. Non-monotonic snapshot isolation: Scalable and strong consistency for geo-replicated transactional systems. In *SRDS*. 163–172.

[2] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '12, London, United Kingdom, June 11-15, 2012*, Peter G. Harrison, Martin F. Arlitt, and Giuliano Casale (Eds.). ACM, 53–64. https://doi.org/10.1145/2254756.2254766

[3] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A critique of ANSI SQL isolation levels. In *ACM SIGMOD Record*,

Vol. 24. ACM, 1–10.

[4] Philip A Bernstein and Nathan Goodman. 1981. Concurrency control in distributed database systems. *Comput. Surveys* 13, 2 (1981), 185–221.

[5] Carsten Binnig, Stefan Hildenbrand, Franz Färber, Donald Kossmann, Juchang Lee, and Norman May. 2014. Distributed snapshot isolation: global transactions pay globally, local transactions pay locally. *The VLDB Journal* 23, 6 (2014), 987–1011.

[6] Andrea Cerone and Alexey Gotsman. 2018. Analysing snapshot isolation. *Journal of the ACM (JACM)* 65, 2 (2018), 11.

[7] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 4.

[8] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *SoCC*. 143–154.

[9] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2013. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.* 31, 3, Article 8 (Aug. 2013), 8:1–8:22 pages.

[10] Transaction Processing Performance Council. 2010. tpc-c benchmark, revision 5.11.

[11] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast remote memory. In *USENIX NSDI*. 401–414.

[12] Jiaqing Du, Sameh Elnikety, and Willy Zwaenepoel. 2013. Clock-SI: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *SRDS*. 173–184.

[13] Sameh Elnikety, Fernando Pedone, and Willy Zwaenepoel. 2005. Database replication using generalized snapshot isolation. In *SRDS*. 73–84.

[14] Alan Fekete, Elizabeth O'Neil, and Patrick O'Neil. 2004. A read-only transaction anomaly under snapshot isolation. *ACM SIGMOD Record* 33, 3 (2004), 12–14.

[15] Lisa Glendenning, Ivan Beschastnikh, Arvind Krishnamurthy, and Thomas Anderson. 2011. Scalable consistency in Scatter. In *SOSP*. 15–28.

[16] Jim Gray and Leslie Lamport. 2006. Consensus on transaction commit. *ACM Transactions on Database Systems (TODS)* 31, 1 (2006), 133–160.

[17] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2016. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided RDMA Datagram RPCs. In *USENIX OSDI*. 185–201.

[18] Masoomeh Javidi Kishi, Sebastiano Peluso, Henry F. Korth, and Roberto Palmieri. 2019. SSS: Scalable Key-Value Store with External Consistent and Abort-free Read-only Transactions. In *ICDCS*. 589–600.

[19] Leslie Lamport. 1998. The Part-Time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (1998), 133–169. https://doi.org/10.1145/279227.279229

[20] Friedemann Mattern et al. 1988. *Virtual time and global states of distributed systems*. Citeseer.

[21] Mohamed Mohamedin, Sebastiano Peluso, Masoomeh Javidi Kishi, Ahmed Hassan, and Roberto Palmieri. 2018. Nemo: NUMA-aware Concurrency Control for Scalable Transactional Memory. In *Proceedings of the 47th International Conference on Parallel Processing, ICPP 2018, Eugene, OR, USA, August 13-16, 2018*. ACM, 38:1–38:10. https://doi.org/10.1145/3225058.3225123

[22] Sebastiano Peluso, Roberto Palmieri, Paolo Romano, Binoy Ravindran, and Francesco Quaglia. 2015. Disjoint-Access Parallelism: Impossibility, Possibility, and Cost of Transactional Memory Implementations. In *PODC*. 217–226.

[23] Sebastiano Peluso, Paolo Romano, and Francesco Quaglia. 2012. SCORe: A Scalable One-Copy Serializable Partial Replication Protocol. In *Middleware 2012*. 456–475.

[24] Sebastiano Peluso, Pedro Ruivo, Paolo Romano, Francesco Quaglia, and Luís Rodrigues. 2016. GMU: Genuine Multiversion Update-Serializable Partial Data Replication. *IEEE Transactions on Parallel and Distributed Systems* 27, 10 (2016), 2911–2925.

[25] Daniel Peng and Frank Dabek. 2010. Large-scale incremental processing using distributed transactions and notifications. (2010).

[26] Maria Pratt and P McElroy. 2001. Oracle9i Replication. *White paper, June* (2001).

[27] Robert Ricci, Eric Eide, and CloudLab Team. 2014. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *; login:: the magazine of USENIX & SAGE* 39, 6 (2014), 36–38.

[28] Yair Sovran, Russell Power, Marcos K Aguilera, and Jinyang Li. 2011. Transactional storage for geo-replicated systems. In *SOSP*. 385–400.

[29] Kimberly L Tripp. 2005. SQL Server 2005 Beta II Snapshot Isolation. (2005).

[30] Juncheng Yang, Yao Yue, and K. V. Rashmi. 2020. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 191–208. https://www.usenix.org/conference/osdi20/presentation/yang