Università
della
Svizzera
italiana

Faculty
of Informatics

USI Technical Report Series in Informatics

RAM-DUR: In-Memory Deferred Update Replication

Daniele Sciascia[1], Fernando Pedone[1]

[1] Faculty of Informatics, Università della Svizzera italiana, Switzerland

Abstract

Many database replication protocols are based on the deferred update replication technique. In deferred update replication, transactions are executed at a single server and broadcast to all servers at commit time. Upon delivering a transaction, each server certifies it to ensure a globally serializable execution. One of the key assumptions of deferred update replication is that servers must store a full copy of the database. Such an assumption is detrimental to performance since in many workloads servers cannot cache the entire database in main memory. This paper introduces RAM-DUR, a variation of deferred update replication whereby transaction execution is in-memory only. RAM-DUR's key insight is a sophisticated distributed cache mechanism that provides high performance and strong consistency without the limitations of existing solutions (e.g., no single server must have enough memory to cache the entire database). In addition to presenting RAM-DUR, we detail its implementation, and provide an extensive analysis of its performance.

Report Info

Published

April 2012

Institution
Faculty of Informatics
Università della Svizzera italiana
Lugano, Switzerland

Online Access
www.inf.usi.ch/techreports

1 Introduction

Deferred update replication (DUR) is a technique to implement highly available data management systems. In brief, servers store a full copy of the database and clients submit transaction requests to a single server. Different clients may choose different servers and the same client may submit different transactions to different servers. A transaction's execution is entirely local to the server chosen to execute it, that is, there is no coordination among servers during the execution of transactions. When a client requests the commit of a transaction, the transaction's updates and some meta data are broadcast to all servers for certification. Certification ensures serializability despite the lack of server coordination during transaction execution (i.e., concurrency control is optimistic [14]). A transaction passes certification and can be committed if it can be serialized with other committed transactions; otherwise the transaction is aborted.

1.1 DUR and other replication techniques

Due to its good performance, several database replication protocols are based on the deferred update replication technique (e.g., [1, 13, 16, 21, 22]). Two main characteristics account for the performance of deferred update replication. First, an update transaction is executed by a single server; the other servers only certify the transaction and apply its updates to their database, should the transaction pass certification; applying a transaction's updates is usually cheaper than executing the transaction. Second, read-only transactions do not need to be certified: A replica can serialize a read-only transaction by carefully synchronizing it locally (e.g., using a multiversion database).

Consequently, deferred update replication is more performance advantageous than other replication techniques, such as primary-backup and state-machine replication. With state-machine replication, every update transaction must be executed by all servers [25]. Thus, adding servers does not increase the through-

put of update transactions; throughput is limited by what one replica can execute. With primary-backup replication [30], the primary first executes update transactions and then propagates their updates to the backups, which apply them without re-executing the transactions; the throughput of update transactions is limited by the capacity of the primary, not by the number of replicas. Servers act as “primaries" in deferred update replication, locally executing transactions and then propagating their updates to the other servers.

1.2 DUR and in-memory transaction execution

When executing transactions, servers strive to store the database in memory (i.e., a cache), thereby avoiding reading from the on-disk database image and improving performance. Since caching the whole database (or a large portion of it) in memory is only possible if the data fits in the server’s memory, some data management architectures recur to partitioning the database across multiple servers in order to increase the chances that the data assigned to each server (i.e., a partition) fits its main memory. Partitioning schemes, however, sometimes restrict transaction execution to a single partition (e.g., [20]) or sacrifice consistency of multi-partition transactions (e.g., [29]).

Partitioning the database across multiple servers without restricting transaction execution and sacrificing consistency is possible with variations of deferred update replication (e.g., [24, 26, 28]). In all such protocols, however, multi-partition transactions have a higher response time than single-partition transactions due to additional communication between partitions during the certification of transactions—essentially, multi-partition transactions have to pass through a voting phase during termination to ensure that they are serializable across the system.

1.3 Main contributions of this paper

This paper introduces in-memory deferred update replication (RAM-DUR), an extension to deferred update replication where the database is partitioned among a set of servers. Ideally, the database will fit the aggregated memory of the compound, but not necessarily the main memory of any individual server. If a server needs a data item it does not cache, instead of retrieving the item from disk, it retrieves it from the cache of a remote server. The rationale behind RAM-DUR is that network access is much faster than a local disk access. Differently from previous approaches, RAM-DUR does not increase the response time of transactions. Transactions are certified following the traditional deferred update replication procedure and there is no voting phase during transaction termination. RAM-DUR’s key insight is a distributed cache mechanism that allows servers to cache portions of the database without sacrificing serializability.

1.4 Roadmap

The remainder of the paper is structured as follows. Section 2 presents our system model and definitions used throughput the paper. Section 3 recalls the deferred update replication approach in detail and discusses its performance. Section 4 introduces in-memory deferred update replication. Section 5 describes our prototype and some optimizations. Section 6 evaluates the performance of the protocol under different conditions. Section 7 reviews related work and Section 8 concludes the paper.

2 System model and definitions

We consider a system composed of an unbounded set $C = \{c_1, c_2, ...\}$ of client processes and a set $S = \{s_1, ..., s_n\}$ of database server processes. We assume a crash-recovery model in which processes can fail by crashing but never perform incorrect actions (i.e., no Byzantine failures). Processes have access to stable storage whose state survives failures. Processes communicate using either one-to-one or one-to-many communication. One-to-one communication uses primitives $send(m)$ and $receive(m)$, where m is a message, and is quasi-reliable: if both the sender and the receiver are correct, then every message sent is eventually received. One-to-many communication relies on atomic broadcast, with primitives $abcast(m)$ and $adeliver(m)$. Atomic broadcast ensures two properties: (1) if message m is delivered by a process, then every correct process eventually delivers m; and (2) no two messages are delivered in different order by their receivers.

The system is *partially synchronous* [10], that is, it is initially asynchronous and eventually becomes synchronous. The time when the system becomes synchronous is called the *Global Stabilization Time (GST)* [10], and it is unknown to the processes. Before GST, there are no bounds on the time it takes for messages to be

transmitted and actions to be executed. After GST, such bounds exist but are unknown. Moreover, in order to ensure liveness, we assume that *after* GST all remaining processes are *correct*—a process that is not correct is *faulty*. A correct process is operational "forever" and can reliably exchange messages with other correct processes. Notice that in practice, "forever" means long enough for some useful computation to be performed.

We assume a multiversion database with data items implemented as tuples $\langle k, v, ts \rangle$, where k is a key, v its value, and ts its version. A transaction is a sequence of read and write operations on data items followed by a commit or an abort operation. We represent a transaction t as a tuple $\langle id, st, rs, ws \rangle$ where id is a unique identifier for t, st is the database snapshot version seen by t, rs is the set of data items read by t, *readset*(t), and ws is the set of data items written by t, *writeset*(t). The readset of t contains the keys of the items read by t; the writeset of t contains both the keys and the values of the items updated by t. The isolation property is *serializability*: every concurrent execution of committed transactions is equivalent to a serial execution involving the same transactions [3].

3 Deferred update replication

In this section we review the deferred update replication technique and discuss some performance issues.

3.1 Deferred update replication

In deferred update replication (DUR), transactions pass through two phases: (1) the *execution phase* and (2) the *termination phase*. The execution phase starts when the client issues the first transaction operation and finishes when the client issues a request to commit or abort the transaction, when the termination phase starts. The termination phase finishes when the transaction is committed or aborted.

In the execution phase of a transaction t, a client c first selects the server s that will execute t's operations; other servers will not be involved in t's execution. When s receives a read command for x from c, it returns the value of x and its corresponding version. The first read determines the *database snapshot* the client will see upon executing other read operations for t. Write operations are locally buffered by c. It is only during transaction termination that updates are propagated to the servers.

In the termination phase, the client atomically broadcasts t's readset and writeset to all servers. Upon delivering t's termination request, s certifies t. Certification ensures a serializable execution; it essentially checks whether t's read operations have seen values that are still up-to-date when t is certified. If t passes certification, then s executes t's writes against the database and assigns each new value the same version number k, reflecting the fact that t is the k-th committed transaction at s.

Database snapshots guarantee that all reads performed by a transaction see a consistent view of the database. Therefore, a read-only transaction t is serialized according to the version of the value t received in its first read operation and does not need certification. Future reads of a transaction return versions consistent with the first read. A read on key k is consistent with snapshot st if it returns the most recent version of k equal to or smaller than st. This rule guarantees that between the version returned for k and st no committed transaction u has modified the value of k (otherwise, u's value would be the most recent one).

Algorithms 1 and 2 illustrate the technique for the client and server, respectively. Primitive retrieve(k, st) (Algorithm 2, line 6) returns from the server's local storage the value of key k either with version equal to st or, if not available, with the greatest version smaller than st. Primitive store(k, v, SC) (Algorithm 2, line 12) includes the specified entry in the server's local storage and sets its version to SC.

3.2 Performance considerations

Deferred update replication assumes that each server stores a full copy of the database. For performance, a server would ideally keep the whole database in main memory, in addition to a durable image stored on disk. Obviously, this is only possible if the database fits in the server's available memory. More often, servers keep a subset of the database in memory (i.e., cached data) and a complete image on disk. Hopefully, transactions will access cached data only; if not, transactions will pay a performance penalty for retrieving entries from the disk.

In Algorithm 2, operations retrieve() and store() abstract these details. A retrieve(k, st) operation may return k's version very quickly from cache or more slowly from the on-disk database image. A store(k, v, SC) operation will typically update k's in-memory entry (if cached), asynchronously update the on-disk database

Algorithm 1 Deferred update replication, client c's code

```
1:  begin(t):
2:      t.st ← ⊥                                  {initially transaction t has no snapshot}
3:      t.rs ← ∅                                  {readset (rs) contains keys read by t, not values}
4:      t.ws ← ∅                                  {writeset (ws) contains keys and values written by t}
5:  read(t, k):
6:      t.rs ← t.rs ∪ {k}                         {add key to readset}
7:      if (k, ⋆) ∈ t.ws then                     {if key previously written...}
8:          return v s.t. (k, v) ∈ t.ws           {return written value}
9:      else                                      {else, if key never written...}
10:         send(read, k, t.st) to some s ∈ S     {send read request}
11:         wait until receive(k, v, st) from s   {wait for response}
12:         if t.st = ⊥ then t.st ← st            {if first read, init snapshot}
13:         return v                              {return value from server}
14: write(t, k, v):
15:     t.ws ← t.ws ∪ {(k, v)}                    {add key to writeset}
16: commit(t):
17:     if t.ws = ∅ then                          {if transaction is read-only...}
18:         return commit                         {commit it right away}
19:     else                                      {else, if it is an update...}
20:         abcast(c, t)                          {abcast it for certification and wait outcome}
21:         wait until receive(outcome) from s ∈ S    {ditto}
22:         return outcome                        {return outcome}
```

image, and synchronously update the database log [11]. Many variations and improvements of this scheme exist. Some recent approaches have suggested storing only the log on disk (i.e., no on-disk database image) and retrieving entries directly from the log [18]. Moreover, this log can be combined with the log implemented by the atomic broadcast primitive (i.e., Paxos) [4, 18]. Asynchronously updating the log is possible if a majority of servers is always operational.

In the next section, we propose an approach in which a subset of servers never accesses disk during the execution and termination of transactions. Instead of reading from disk entries missing in the in-memory storage, such a server retrieves them from the in-memory storage of a remote server. Moreover, committed updates are only applied to the keys currently stored in main memory.

4 In-memory deferred update replication

In this section, we first outline the algorithm for in-memory deferred update replication (RAM-DUR) and then present a complete protocol. We formally argue about RAM-DUR correctness in the Appendix.

4.1 General idea

RAM-DUR distinguishes between two types of servers: *core servers*, which execute the traditional deferred update replication, and *volatile servers* (*vnodes*), which store a subset of the database in memory only. The purpose of vnodes is to increase the performance of transaction execution by ensuring that data is always in memory. The system does not rely on vnodes for durability. Vnodes essentially implement a caching layer that ensures consistent execution of transactions. As we discuss in Section 4.3, transaction durability is guaranteed by the core servers.

RAM-DUR partitions database entries among vnodes based on their key, using range or hash partitioning. A vnode is the *owner* of all entries assigned to it as a result of the partitioning scheme. A vnode executes both read operations from the clients and *remote read* operations from other vnodes. A vnode v issues a remote read for key k to the owner of k when some client requests to read k and v does not store k locally (e.g., v is not the owner of k). In addition to its assigned entries, vnodes can cache any other entries, as long as memory is available. A vnode is required to store new versions of the keys it owns, but it may discard other entries, as long as consistency is not violated.

We define two operations to manipulate the local storage of vnodes: (1) *lookup*(k, st), which returns a

Algorithm 2 Deferred update replication, server s's code

```
 1: Initialization:
 2:    SC ← 0                                   {snapshot counter: last snapshot created by s}
 3:    WS[...] ← ∅                              {ws vector: one entry per committed transaction}
 4: when receive(read, k, st) from c
 5:    if st = ⊥ then st ← SC                   {if first read, initialize snapshot}
 6:    v ← retrieve(k, st)                      {most recent version ≤ st in database}
 7:    send(k, v, st) to c                      {return result to client}
 8: when adeliver(c, t)
 9:    outcome ← certify(t)                     {outcome is either commit or abort}
10:    if outcome = commit then                 {if it passes certification...}
11:       for all (k, v) ∈ t.ws do              {for each update in t's writeset...}
12:          store(k, v, SC)                    {...apply update to database}
13:    send(outcome) to c                       {return outcome to client}
14: function certify(t)                         {used in line 9}
15:    for i ← t.st to SC do                    {for all concurrent transactions...}
16:       if WS[i] ∩ t.rs ≠ ∅ then              {if some intersection...}
17:          return abort                       {transaction must abort}
18:    SC ← SC + 1                              {here no intersection: one more snapshot}
19:    WS[SC] ← items(t.ws)                     {keep track of committed writeset}
20:    return commit                            {transaction must commit}
```

tuple ⟨*key*, *value*, *version*⟩ from the local storage of the vnode or from the owner of k (by remotely fetching the tuple); and (2) *apply*(k, v, st), which stores item (k, v) with version st in the vnode's storage. The lookup request must return the largest version of k equal to or smaller than v.

Figure 1 depicts a simplified view of the various storage abstractions of a vnode. Clients (i.e., applications) have access to data through the *read* and *write* primitives, which are implemented on top of *lookup* and *apply* primitives. A *lookup* will be translated into a local *retrieve* operation or a *remote-read* operation, depending on whether the item read is cached or not, respectively; *apply* is implemented by a call to *store*. The *remote-read* request is submitted to the owner of the missing entry, which will translate it into a *lookup* to its local storage.

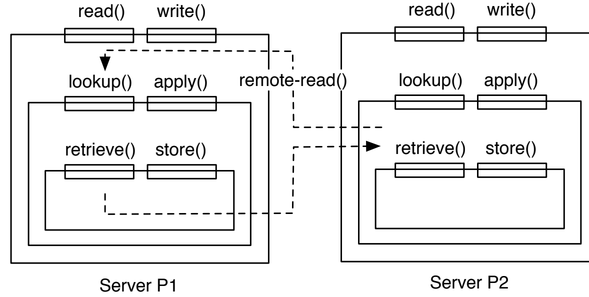

Figure 1: Simplified storage abstractions of a vnode.

In order to preserve serializability, *lookup* and *apply* have to follow a number of rules. We introduce these rules next and justify their need with problematic executions they avoid. In all executions described next, vnode P_1 is the owner of data item x and vnode P_2 does not own x and does not have a cached version of x. Transaction t issues read operations on x against P_2.

Execution 1. Transaction t issues a read operation on x for version 11 against P_2 (see Figure 2(a)). As a result, P_2 issues a remote read operation against P_1. When P_1 handles the request, its *SC* is 10 while P_2 had delivered snapshot 11 before starting t (recall from Algorithm 2 that *SC* is the latest snapshot created by a server). P_1 returns the largest version of x it knows about equal to or smaller then 11, which in this case is version 10. However, snapshot 11 includes an update to x, which should be visible to transaction t.

To avoid this case, we introduce Rule 1:

- *Rule 1. A vnode can only reply to a remote read for version v if v is less than or equal to the vnode's SC.*

Execution 2. In this case (Figure 2(b)), t issues a read operation for key x with version 10 against P_2. P_2 sends a remote read request to P_1 and then delivers snapshot 11, before receiving a reply from P_1. If snapshot 11 contains an updated version of x, P_2 is allowed to discard it (because P_2 is not the owner of x). Later P_2 receives a reply for x with version 10. To avoid future requests to x, P_2 caches x with version 10 locally. Suppose a later transaction t' reads x with version 11, P_2 returns x with version 10, i.e. the most recent x with version less than or equal to 11. This execution is not serializable because t' should see version 11 of x.

This case is excluded by Rule 2:

- *Rule 2. A vnode must not discard a version of k that is delivered if it has a pending remote request for k.*

Execution 3. Assume that P_2 is ahead in the execution by two snapshots (Figure 2(c)). P_2 requests version 8 of key x. When P_1 receives the request it sends key x at version 8 to P_1. According to the rules defined so far, P_2 can cache $(x, 8)$ locally. However, a later transaction t with $st = 9$ might request $(x, 9)$, and because P_2 is ahead in the execution it already discarded version 9 and returns $(x, 8)$.

Rule 3 avoids this case.

- *Rule 3. A key can only be cached if the SC of the vnode handling a request for k is bigger than or equal to the SC of the requesting vnode. Also, k can be cached only if it is the newest version smaller than SC.*

Execution 4. Figure 2(d) shows an execution in which both P_1 and P_2 deliver snapshot 10. Right after delivering the snapshot, P_1 garbage collects item x with version 10. Later transaction t requests to read item x with version 10, and accordingly P_2 sends a remote request to P_1. Since version 10 of x was garbage collected, P_1 returns the most recent version less than 10, namely x with version 9.

Rule 4 applies to both cached and non-cached entries.

- *Rule 4. A vnode can garbage collect version v of item k only if it has already garbage collected all versions of k with version less than v. The owner of k must preserve at least one version (the latest one), while vnodes caching k can remove all versions.*

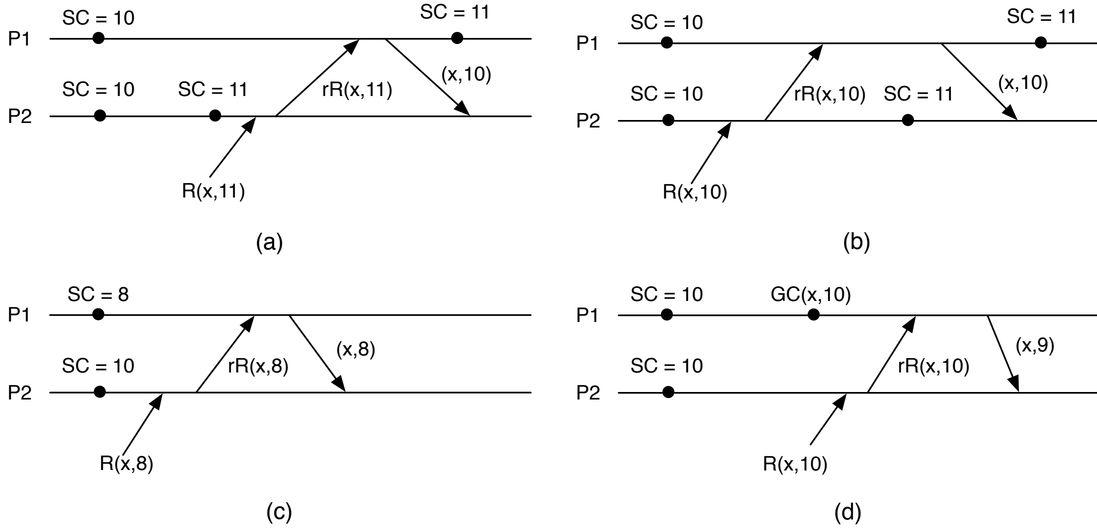

Figure 2: Problematic executions involving vnodes P_1 and P_2 in naive use of lookup and apply.
SC: snapshot counter. $R(x, i)$: read request for version i of data item x. $rR(x, i)$: remote read request for version i of data item x. $GC(x, i)$: garbage collect data item x with version i.

4.2 Algorithm in detail

M-DUR is an extension to the deferred update replication that does not require modifications to the client side. In this section we only discuss the protocol for vnodes, shown in Algorithm 3, while we assume clients follow Algorithm 1.

Algorithm 3 RAM-DUR, server s's code

```
1:  Initialization:
2:    SC ← 0                                              {initialize snapshot counter}
3:    WS[...] ← ∅                                         {initialize committed writesets}
4:  when receive(read, k, st) from c
5:    if st = ⊥ then st ← SC                              {if first read, initialize snapshot}
6:    v ← lookup(k, st)                                   {most recent version ≤ st in database}
7:    send(k, v, st) to c                                 {return result to client}
8:  when adeliver(c, t)
9:    outcome ← certify(t)                                {outcome is either commit or abort}
10:   if outcome = commit then                            {it it passes certification...}
11:     for all (k, v) ∈ t.ws do                          {for each update in t's writeset...}
12:       apply(k, v)                                     {...apply update to database}
13:   send(outcome) to c                                  {return outcome to client}
14: when receive(remote-read, k, sc, st) from r and
                              SC ≥ st                     {Rule 1}
15:   v ← lookup(k, st)
16:   cacheable ← false
17:   if v is newest version stored locally and
                              SC ≥ sc then                {Rule 3}
18:     cacheable ← true
19:   send(k, v, st, cacheable) to r
20: function lookup(k, st)                                {used in lines 6 and 15}
21:   if k is stored locally then
22:     return retrieve(k, st)
23:   else
24:     send(remote-read, k, SC, st) to owner(k)
25:     wait until received (k, v, st, cacheable)
26:     if cacheable then store(k, v, st)
27:     return v
28: function apply(k, v)                                  {used in line 7}
29:   if s = owner(k) or k is stored locally or
                 pending remote request for k then        {Rule 2}
30:     store(k, v, SC)
31: function certify(t)                                   {used in line 4}
32:   unmodified from Algorithm 2
```

Read requests are executed at vnodes by the lookup() primitive (line 6), which essentially either returns the entry stored locally (line 22) or requests the entry to the key owner (lines 24 and 25). According to Rule 3, the returned entry is cacheable if it is the newest entry stored by its owner and the owner's *SC* is at least as big as the requested version (line 17). Before responding to a remote read request, the owner of a key makes sure that its *SC* is equal to or greater than the requested version, ensuring Rule 1 (line 14).

Upon delivering a committing transaction t (line 8), a vnode certifies t using the same procedure as core servers (line 9, 31, and 32). If the outcome of certification is commit, t's modification to the database are applied to the local storage (lines 11 and 12). The apply primitive only keeps an entry in local storage if it satisfies Rule 2 (lines 29 and 30).

4.3 Discussion

The motivation for vnodes in RAM-DUR is performance. Two mechanisms implemented by vnodes improve performance. First, by requesting a missing data from a remote vnode, no local disk reads are necessary. This approach is quite effective since retrieving a missing item from the main memory of a remote vnode is one order of magnitude faster than retrieving the item from the local disk. Second, items often accessed are cached by vnodes, thus reducing network traffic and execution delay. Cached data is updated as part of the termination of transactions, ensuring "data freshness".

Our mechanism to broadcast terminating transactions to servers is based on Paxos [15]. Paxos's "accept-

ors" can be co-located with core servers or deployed at independent nodes, the approach used in all our DUR and RAM-DUR experiments. Therefore, when a transaction is delivered by a server, the transaction's contents and order have been safely stored by the acceptors and will not be forgotten, despite failures. This mechanism ensures transaction durability despite the crash of vnodes.

Although recovering a vnode from the state of core servers is a simple operation, until it recovers, all entries it owns will be unavailable for other vnodes. To prevent blocking due to the crash of a vnode, we extend the logic of vnodes to request missing items from core servers, should they suspect the crash of one of its peers. Notice that as long as core servers respect the rules presented in the previous sections, consistency will be preserved.

5 Implementation and optimizations

For the implementation of RAM-DUR, we use Ring Paxos [19] as our atomic broadcast primitive. Acceptors log delivered values on disk asynchronously, as part of the atomic broadcast execution—we assume that there is always a majority of operational acceptors in order to ensure durability. Optionally, RAM-DUR servers can store a consistent snapshot of their in-memory state to disk. This checkpointing mechanism can be used for backups, or to restart the whole system by replaying only the tail of the log since the last checkpoint.

Our DUR and RAM-DUR prototypes broadcast transactions in small batches. This is essentially the well-known *group commit* optimization in centralized databases. In our case, it amortizes the cost of the atomic broadcast primitive over several transactions.

We use bloom filters to efficiently check for intersections between transaction readsets and writesets. The implementation keeps track of only the past K writeset bloom filters, where K is a configurable parameter of the system. There are two more advantages in using bloom filters: (1) bloom filters have negligible memory requirements; and (2) they allow us to send just the hashes of the readset when broadcasting a transaction, thus reducing network bandwidth. However, using bloom filters results in a negligible number of transactions aborted due to false positives.

We used the same code base to implement standard DUR. In the case of DUR, we disabled those features that are unique to RAM-DUR (i.e., each server has a full copy of the database, does not issue remote requests, and has no cache mechanism). Local storage of DUR is implemented using Berkeley DB (BDB); for RAM-DUR, we provide an alternative storage implemented as a multiversion hash-table, optimized for keeping data in-memory only.

6 Performance evaluation

We assess next the performance of RAM-DUR. We measured throughput and latency of RAM-DUR and compared them against standard DUR, under workloads that both fit and don't fit in the memory of a single server.

6.1 Setup and benchmarks

We ran the experiments in a cluster of Dell SC1435 servers equipped with two dual-core AMD-Opteron 2.0 GHz processors and 4 GB of main memory, and interconnected through an HP ProCurve2900-48G Gigabit Ethernet switch. Servers are attached to a 73 GB 15krpm SAS hard-disk.

We evaluated the performance of RAM-DUR using a simple micro-benchmark. The benchmark consists of two types of transactions (see Table 1): (1) *update transactions* perform a read and a write on the same key; (2) *read-only transactions* perform two read operations on different keys. Keys are 4 bytes long, while values consist of 1024 bytes.

Clients are evenly distributed across servers. Each client issues a mix of 10% update and 90% read-only transactions. Keys are selected randomly over a portion large $1/n$ of the total number of items in the dataset, where n is the number of servers. Clients connected to the same vnode access the same portion, which ensures some data locality.

We consider two datasets (see Table 2): (1) a *small database*, where servers are loaded with 100 thousand data items per vnode and the dataset fits in the memory of a single server; and (2) a *large database*, where servers are loaded with 400 thousand items per vnode and the dataset does no fit in the memory of a single server, but it fits the aggregated memory of all servers.

Moreover, to make the comparison between DUR and RAM-DUR fair, we use the same number of nodes. Therefore, in DUR and RAM-DUR 3, only three servers in total execute transactions, core servers in DUR and

vnodes in RAM-DUR. We assess the benefit of additional vnodes with RAM-DUR 6, a configuration with 6 vnodes.

Type	**Operations**	**Frequency**
Read-only	2 reads	90%
Update	1 read, 1 write	10%

Table 1: Transaction types in workload

Dataset	**Size**	**Characteristic**
Small DB	100K items per server	fits RAM of a single server
Large DB	400K items per server	fits aggregated servers' RAM

Table 2: Datasets in workload

6.2 Throughput and latency

Figure 3 shows throughput and latency of DUR and RAM-DUR under maximum load. When data fits in memory (i.e., Small DB) we notice that both DUR and RAM-DUR perform well, although RAM-DUR performs two times faster than DUR. We attribute this difference to the fact that DUR is fully replicated (and thus, every update must be applied to storage). Also, RAM-DUR's in-memory only storage is faster than DUR's Berkeley DB-based storage. In terms of latency, DUR does better than RAM-DUR, especially for read-only transactions. This difference is due to the fact that RAM-DUR sustains more clients (in this experiment we run 8 and 32 clients per server for respectively DUR and RAM-DUR).

When the dataset does not fit in the memory of a single node (i.e., Large DB) we notice a significant impact on DUR's performance. Performing reads from the local storage and applying updates requires disk access. In this setup, DUR performs only 94 update transactions and 816 read-only transactions per second. On the other hand, we observed a minimal performance impact in the case of RAM-DUR. DUR's latency is also heavily impacted: latency of update transactions more than doubled; latency of read-only transactions went from only 0.4 milliseconds to slightly less than 6 milliseconds.

Finally, increasing the number of servers in RAM-DUR is beneficial. Going from 3 to 6 vnodes roughly doubled the performance of RAM-DUR, at the same time keeping latency low.

6.3 Performance under remote requests

In the following experiment, we show how quickly a vnode builds its working set and how its performance compares to a core server in DUR, where remote requests never take place. In this experiment, an equal number of clients issue transactions against DUR and RAM-DUR and, for the sake of the comparison, we only consider the case where the dataset fits in memory. Figure 4 shows throughput, latency, and the number of remote requests over time. At time 0, RAM-DUR servers where loaded with 100 thousand items per node, however they store locally only data items they own, and do not store any cached items. Transaction execution starts at time 1, and at this time RAM-DUR's throughput and latency are worse than DUR's. As the number of remote requests diminish, RAM-DUR quickly catches up with DUR, and at time 5 both protocols roughly perform the same. At time 10 the number of remote requests per second is one fifth of the value in the beginning of the execution, and RAM-DUR starts approaching its peak performance.

6.4 Cache-only vnodes

We evaluate next the effects of online additions of cache-only vnodes (Figure 5). A cache-only vnode is a vnode which does not own any data items. Initially there are 6 vnodes. At time 20, a cache-only vnode is added to the compound and at time 60 another one is included. System throughput increases as more vnodes are added. The first graph, on the top of Figure 5, shows the aggregated throughput of all vnodes in the system as well as the additional throughput added by the 7th vnode. In terms of latency, we observe that the addition of a new vnode slightly increases the average aggregated latency. This effect is due to the fact that cache-only vnodes start without any items, and therefore every read results in a remote request initially.

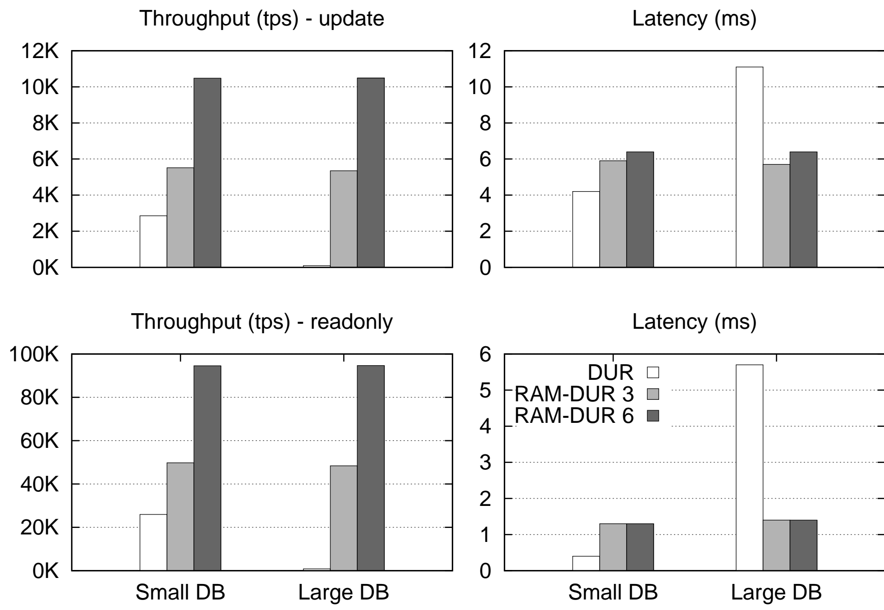

Figure 3: Throughput and latency.

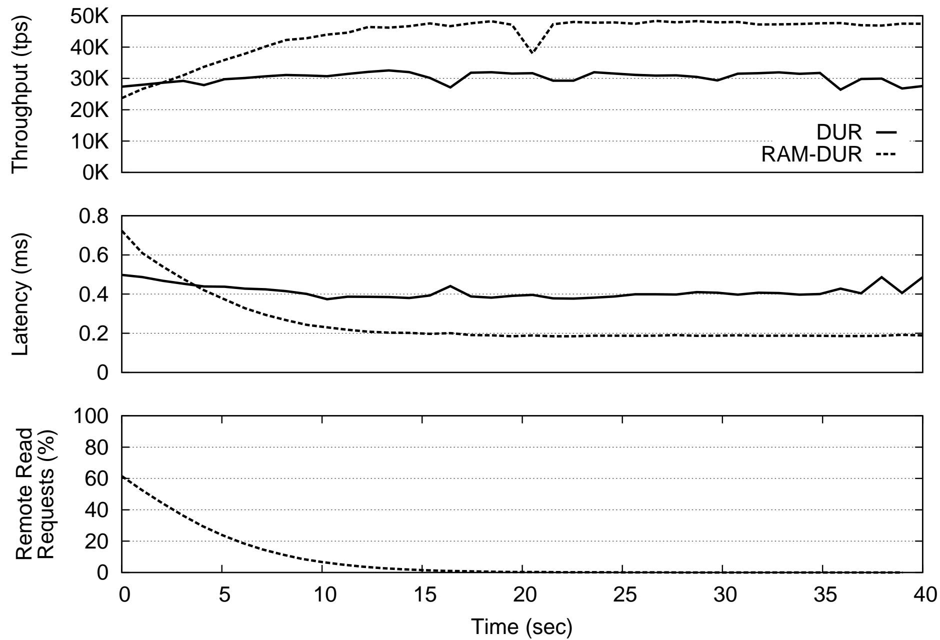

Figure 4: Performance under remote requests (Small DB).

The increase in latency is experienced only at the added vnode. However as new vnodes cache their working set, remote requests decrease. After 10–15 seconds, the 7th vnode contributes the same throughput as the vnodes that were already present at the beginning of the execution, its latency approaches the aggregated average latency, and the number of remote requests per second approaches zero.

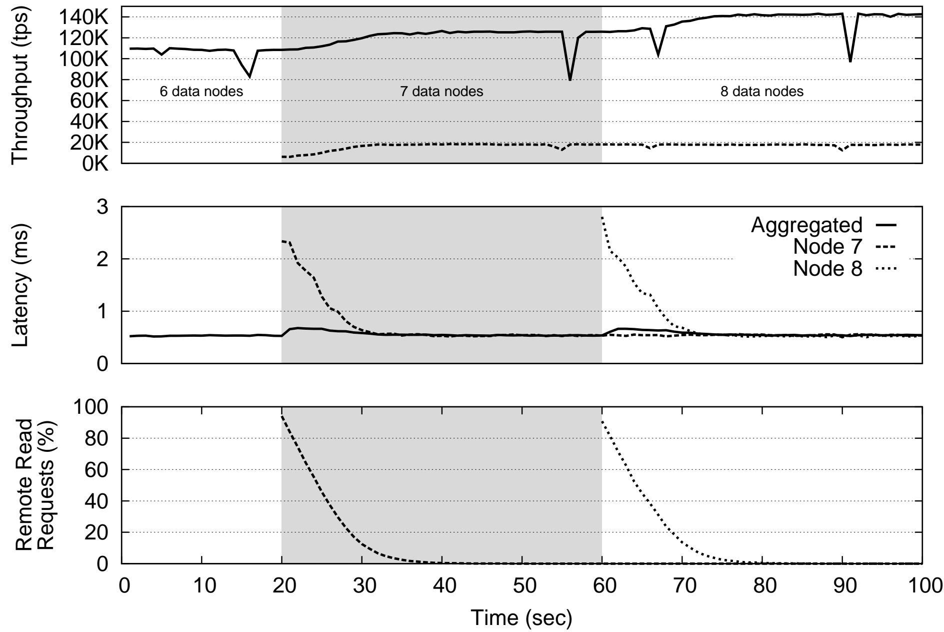

Figure 5: Adding cache-only vnodes (Large DB).

7 Related work

A number of protocols for deferred update replication where servers keep a full copy of the database have been proposed (e.g., [1, 13, 16, 21, 22]). Similar to RAM-DUR some protocols provide partial replication (e.g., [23, 27, 28]) to improve the performance of deferred update replication. This also improves scalability in that only the subset of servers addressed by a transaction applies updates to their local database. These protocols, including RAM-DUR, require transactions to be atomically broadcast to all participants. However, only RAM-DUR addresses the case where data is kept always in memory.

Full database replication hinders the scalability of update transactions, which in RAM-DUR is inherently limited by the number of transactions that can be ordered. In previous work we addressed the problem of scalability in the context of deferred update replication [26]. Nothing prevents combining RAM-DUR's mechanisms described here with other techniques for improving scalability.

G-Store [8] proposes a *key group protocol* that allows transactional multi-key access over dynamic and non-overlapping groups of keys. To execute a transaction that accesses multiple keys, the key group protocol must transfer ownership for all keys in a group to a single node. Once a node owns a group, it can efficiently execute multi-key transactions. RAM-DUR has no such constraints.

H-Store [12] shares some design considerations with RAM-DUR, although the main approach is different. In H-Store the execution is optimized depending on transaction classes and schema characteristics in order to distinguish *two-phase*, *strictly two-phase* and *sterile* transactions and attain high performance. This can be done automatically, or by pre-declaring transaction classes, which makes the system less flexible.

Dynamo [9] is a distributed key-value store developed and in use at Amazon. Cassandra [5] and Voldemort [31] are open-source projects inspired by Dynamo. RAM-DUR and Dynamo share a similar query interface, but their semantics differ. First, Dynamo does not support transactional operations. Second, it is based on *eventual consistency*, where operations are never aborted but isolation is not guaranteed. Conflict resolution is moved to the client, meaning that clients need to handle conflicts by reconciling conflicting versions of data items. This mechanism ensures high availability at the cost of transparency.

COPS [17] is a wide-area storage system that ensures a stronger version of causal consistency, which in addition to ordering causally related write operations also orders writes on the same data items. Walter [29] offers an isolation property called Parallel Snapshot Isolation (PSI) for databases replicated across multiple

data centers. PSI guarantees snapshot isolation and total order of updates within a site, but only causal ordering across data centers.

Differently from previous works, Sinfonia [2] offers stronger guarantees by means of minitransactions on unstructured data. Similarly to RAM-DUR, minitransactions are certified upon commit. Differently from RAM-DUR, both update and read-only transactions must be certified in Sinfonia, and therefore can abort. Read-only transactions do not abort in RAM-DUR.

Google's Bigtable [6] and Yahoo's Pnuts [7] are distributed databases that offer a simple relational model (e.g., no joins). Bigtable supports very large tables and copes with workloads that range from throughput-oriented batch processing to latency-sensitive applications. Pnuts provides a richer relational model than Bigtable: it supports high-level constructs such as range queries with predicates, secondary indexes, materialized views, and the ability to create multiple tables. However, none of these databases offer full transactional support.

8 Conclusion

This paper proposes an extension to deferred update replication. Deferred update replication is widely used by several database protocols due to its performance advantages: scalable read-only transactions, and good throughput under update transactions. RAM-DUR extends deferred update replication targeting in-memory execution. We introduce two mechanisms, remote reads and caching, which allow us to significantly speedup the execution phase in workloads that do not fit the memory of a single server. Moreover, we do so without sacrificing consistency. We assessed the performance of RAM-DUR under different scenarios and showed how RAM-DUR can quickly add cache-only vnodes to further improve system throughput online.

References

[1] D. Agrawal, G. Alonso, A. El Abbadi, and I. Stanoi. Exploiting atomic broadcast in replicated databases. In *Proceedings of EuroPar*, 1997.

[2] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 159–174, 2007.

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

[4] L. Camargos, F. Pedone, and M. Wieloch. Sprint: a middleware for high-performance transaction processing. In *Proceedings of the 2nd ACM European Conference on Computer Systems*, EuroSys '07, pages 385–398. ACM, 2007.

[5] http://cassandra.apache.org/.

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

[7] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, 2008.

[8] S. Das, D. Agrawal, and A. El Abbadi. G-store: a scalable data store for transactional multi key access in the cloud. In *SoCC '10: Proceedings of the 1st ACM symposium on Cloud computing*, New York, 2010. ACM.

[9] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, 2007.

[10] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

[11] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2 edition, 2008.

[12] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.*, 1(2):1496–1499, 2008.

[13] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-r, a new way to implement database replication. In *Proceedings of 26th International Conference on Very Large Data Bases (VLDB'2000)*, Cairo, Egypt, 2000.

[14] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, 1981.

[15] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[16] Y. Lin, B. Kemme, M. Patino-Martinez, and R. Jimenez-Peris. Middleware based data replication providing snapshot isolation. In *International Conference on Management of Data (SIGMOD)*, Baltimore, Maryland, USA, 2005.

[17] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, 2011.

[18] D. Malkhi, M. Balakrishnan, J. Davis, V. Prabhakaran, and T. Wobber. From paxos to corfu: a flash-speed shared log. *SIGOPS Oper. Syst. Rev.*, 46(1):47–51, February 2012.

[19] P. J. Marandi, M. Primi, N. Schiper, and F. Pedone. Ring Paxos: A high-throughput atomic broadcast protocol. In *International Conference on Dependable Systems and Networks (DSN)*, 2010.

[20] http://memcachedb.org/.

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

[22] F. Pedone, R. Guerraoui, and A. Schiper. The Database State Machine approach. *Distrib. Parallel Databases*, 14:71–98, July 2003.

[23] N. Schiper, R. Schmidt, and F. Pedone. Optimistic algorithms for partial database replication. In *Proceedings of the 10th International Conference on Principles of Distributed Systems (OPODIS)*, pages 81–93. Springer, 2006.

[24] N. Schiper, P. Sutra, and F. Pedone. P-store: Genuine partial replication in wide area networks. In *Proceedings of the 29th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 214–224. IEEE, 2010.

[25] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[26] D. Sciascia, F. Pedone, and F. Junqueira. Scalable deferred update replication. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2012.

[27] D. Serrano, M. Patino-Martinez, R. Jiménez-Peris, and B. Kemme. Boosting database replication scalability through partial replication and 1-copy-snapshot-isolation. In *Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 290–297. IEEE, 2007.

[28] A. Sousa, F. Pedone, R. Oliveira, and F. Moura. Partial replication in the database state machine. In *Proceedings of the 1st International Symposium on Network Computing and Applications (NCA)*, pages 298–309. IEEE, 2001.

[29] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, pages 385–400, 2011.

[30] M. Stonebraker. Concurrency control and consistency of multiple copies of data in distributed Ingres. *IEEE Transactions on Software Engineering*, SE-5:188–194, 1979.

[31] http://project-voldemort.com/.

Appendix

We show next that RAM-DUR only produces serializable schedules. In the proof, we argue that every history h produced by RAM-DUR has an acyclic multi-version serialization graph (MVSG) [3]; if $MVSG(h)$ is acyclic, then h is view equivalent to some serial execution of the transactions in h [3].

We first prove the following property about RAM-DUR.

Proposition 1 *(No holes.) If a lookup*(k, st) *request results in tuple* $\langle k,v,x\rangle$*, where* $x \le st$*, then no committed transaction creates tuple* $\langle k,u,y\rangle$ *such that* $x < y \le st$*.*

Proof: Assume for a contradiction that at server s, a lookup(k, st) returns $\langle k,v,x\rangle$, $x \le st$, and there is a transaction t that creates entry $\langle k,u,y\rangle$ and $x < y \le st$.

From Algorithm 3, there are two cases to be considered:

Case (a): $\langle k,v,x\rangle$ *is stored locally.* When s issues a retrieve(k, st) request, it returns the most recent version smaller than or equal to st from the store; thus, entry $\langle k,u,y\rangle$ must not be in the store when s executes the retrieve operation. But when s executes retrieve(k, st), it has already certified every transaction that creates snapshot $y \le st$.

We divide case (a) in three sub-cases (Rule 2):

- *Case (a.1)* s is the owner of k. Then it must store every update to k, including version y. Thus, we conclude that s is not k's owner.
- *Case (a.2)* s caches entry $\langle k,v,x\rangle$. Then either (a.2.1) s received from r, k's owner, cacheable entry $\langle k,v,x\rangle$ or (a.2.2) s cached an earlier version of k and delivered update $\langle k,v,x\rangle$. In case (a.2.1), by the algorithm, when r replied to the remote read request (*remote-read*$, k, SC_s, st)$ from s, v was the newest version stored locally and $SC_r \ge SC_s$ (Rule 3). It follows that there is no version y of k such that $x < y \le SC_s$, a contradiction since $st \le SC_s$. Case (a.2.2) is similar to (a.1).
- *Case (a.3)* s garbage collected $\langle k,u,y\rangle$. A contradiction to Rule 4, because there exists older version x of k which was not garbage collected.

Case (b): $\langle k,v,x\rangle$ *is not stored locally.* Therefore, s sends message (*remote-read*$, k, SC_s, st)$ to server r, the owner of k, and it must be that $st \le SC_s$. Server r only executes s's request when $SC_r \ge st$ (Rule 1). Hence, r returns $\langle k,u,y\rangle$, and not version x, a contradiction that concludes the proof. □

Theorem 1 *RAM-DUR guarantees serializability.*

Proof: MVSG is a directed graph, in which the nodes represent committed transactions. There are three types of directed edges in MVSG: (a) read-from edges, (b) version-order edges type I, and (c) version-order edges type II. These types are described below. We initially consider update transactions only, i.e., edges that connect two update transactions. Then we consider read-only transactions.

Update transactions. From the algorithm, the commit order of transactions induces a version order on every data item: if transactions t_i and t_j create entries $\langle k, v_i, ts_i\rangle$ and $\langle k, v_j, ts_j\rangle$, respectively, then $SC(t_i) < SC(t_j) \Leftrightarrow ts_i < ts_j$, where $SC(t)$ is the snapshot counter associated with transaction t and corresponds to its commit order.

To show that $MVSG(h)$ has no cycles, we prove that for every edge $t_i \to t_j$ in $MVSG(h)$, it follows that $SC(t_i) < SC(t_j)$. The proof continues by considering each edge type in $MVSG(h)$.

1- Read-from edge. If t_j reads $\langle k, v_i, ts_i\rangle$ from t_i, then $t_i \to t_j \in MVSG(h)$.

We have to show that $SC(t_i) < SC(t_j)$. From the algorithm, $ts_i = SC(t_i)$, which is the value of global counter SC at server s when t_i was certified at s. Since transactions only read committed data from other transactions, and t_j reads an entry from t_i, t_j is certified after t_i is certified. For each transaction that passes certification, s increments SC, and thus, it must be that $SC(t_i) < SC(t_j)$.

2- Version-order edge type I. If t_i and t_j create entries $\langle k, v_i, ts_i\rangle$ and $\langle k, v_j, ts_j\rangle$, respectively, such that $ts_i < ts_j$, then $t_i \to t_j \in MVSG(h)$.

Since the commit order induces the version order, we have that $ts_i < ts_j \Leftrightarrow SC(t_i) < SC(t_j)$.

3- Version-order edge type II. If t_i reads $\langle k, v_k, ts_k\rangle$ from t_k and t_j creates entry $\langle k, v_j, ts_j\rangle$ such that $ts_k < ts_j$, then $t_i \to t_j \in MVSG(h)$.

Since no two transactions have the same commit timestamp, either $SC(t_i) < SC(t_j)$ (i.e., what we must show), or $SC(t_j) < SC(t_i)$. For a contradiction, assume the latter, which together with $ts_k < ts_j$ leads to (a) $SC(t_k) < SC(t_j) < SC(t_i)$. Since t_i reads from t_k, it follows that (b) $SC(t_k) \le ST(t_i)$, where $ST(t)$ is the database snapshot version seen by t.

From (a), (b), and the fact that $ST(t_i) < SC(t_i)$ (i.e., the database snapshot version seen by t_i must precede the version t_i creates), we have to show that the cases that follow cannot happen (see Figure 6):

Case (i): $SC(t_k) \leq ST(t_i) < SC(t_j) < SC(t_i)$. In this case, t_j's writeset must be considered when certifying t_i since t_j commits before t_i is certified. t_i can only commit if its readset does not intersect t_j's writeset, but since $t_i.rs \cap t_j.ws = \{k\}$, t_i fails certification, a contradiction.

Case (ii): $SC(t_k) < SC(t_j) < ST(t_i) < SC(t_i)$. When t_i reads key k, it receives $\langle k, v_k, SC(t_k)\rangle$ from the storage, and not the value created by t_j. The read operation is translated into a lookup$(k, ST(t_i))$ and returns $\langle k, v_k, SC(t_k)\rangle$, where $SC(t_k) < ST(t_i)$. Thus, from Proposition 1 (no holes), there is no transaction that creates entry $\langle k, -, SC(t_j)\rangle$, a contradiction.

Read-only transactions. Let t_q be a read-only transaction in h. Since t_q does not update any data item, any edge involving t_q in $MVSG(h)$ is of the type either (a) read-from edge: $t_i \rightarrow t_q$ or (b) version-order type II: $t_q \rightarrow t_j$, where t_i and t_j are update transactions. We show that the former implies $SC(t_i) < ST(t_q)$ and the latter implies $ST(t_q) < SC(t_j)$.

4- Read-from edge. Since $t_i \rightarrow t_q \in MVSG(h)$, t_q must read some data item written by t_i. Since only committed versions can be read, it follows that $SC(t_i) < ST(t_q)$.

5- Version-order edge type II. Since $t_q \rightarrow t_j \in MVSG(h)$, there must exist some transaction t_k such that both t_k and t_j create entries $\langle k, v_k, ts_k\rangle$ and $\langle k, v_j, ts_j\rangle$, where $ts_k < ts_j$, and t_q reads $\langle k, v_k, ts_k\rangle$. By an argument similar to case (3) above, it must be that $ST(t_q) < SC(t_j)$.

The proof continues by contradiction: assume t_q is involved in a cycle c in $MVSG(h)$. Then, for each edge $t_a \rightarrow t_b \in c$, $SC(t_a) < SC(t_b)$ if both t_a and t_b are update transactions (from the first part of the proof), $SC(t_a) < ST(t_b)$ if t_b is a read-only transaction (from case (4) above), and $ST(t_a) < SC(t_b)$ if t_a is a read-only transaction (from case (5) above). Thus if c exists, it follows that $ST(t_q) < ST(t_q)$, a contradiction that concludes the proof. □

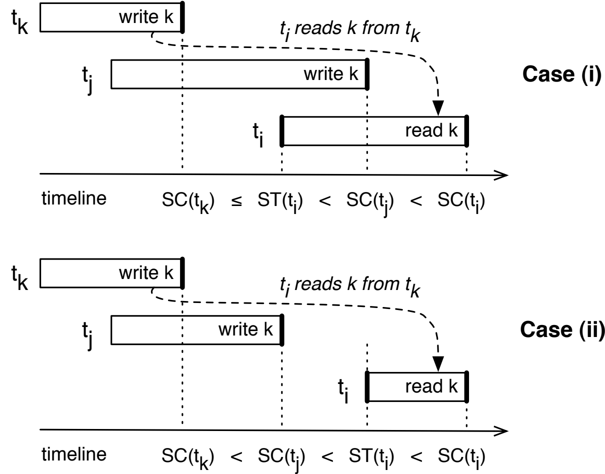

Figure 6: Instances of cases (i) and (ii) in proof.