# Scalable Atomic Visibility with RAMP Transactions

Peter Bailis, Alan Fekete†, Ali Ghodsi, Joseph M. Hellerstein, Ion Stoica
UC Berkeley and †University of Sydney

## ABSTRACT

Databases can provide scalability by partitioning data across several servers. However, multi-partition, multi-operation transactional access is often expensive, employing coordination-intensive locking, validation, or scheduling mechanisms. Accordingly, many real-world systems avoid mechanisms that provide useful semantics for multi-partition operations. This leads to incorrect behavior for a large class of applications including secondary indexing, foreign key enforcement, and materialized view maintenance. In this work, we identify a new isolation model—Read Atomic (RA) isolation—that matches the requirements of these use cases by ensuring *atomic visibility*: either all or none of each transaction's updates are observed by other transactions. We present algorithms for Read Atomic Multi-Partition (RAMP) transactions that enforce atomic visibility while offering excellent scalability, guaranteed commit despite partial failures (via *synchronization independence*), and minimized communication between servers (via *partition independence*). These RAMP transactions correctly mediate atomic visibility of updates and provide readers with snapshot access to database state by using limited multi-versioning and by allowing clients to independently resolve non-atomic reads. We demonstrate that, in contrast with existing algorithms, RAMP transactions incur limited overhead—even under high contention—and scale linearly to 100 servers.

## 1. INTRODUCTION

Faced with growing amounts of data and unprecedented query volume, distributed databases increasingly split their data across multiple servers, or *partitions*, such that no one partition contains an entire copy of the database [7, 13, 18, 19, 22, 29, 43]. This strategy succeeds in allowing near-unlimited scalability for operations that access single partitions. However, operations that access multiple partitions must communicate across servers—often synchronously—in order to provide correct behavior. Designing systems and algorithms that tolerate these communication delays is a difficult task but is key to maintaining scalability [17, 28, 29, 35].

In this work, we address a largely underserved class of applications requiring multi-partition, atomically visible[1] *transactional* access: cases where all or none of each transaction's effects should be visible. The status quo for these multi-partition atomic transactions provides an uncomfortable choice between algorithms that are fast but deliver inconsistent results and algorithms that deliver consistent results but are often slow and unavailable under failure. Many of the largest modern, real-world systems opt for protocols that guarantee fast and scalable operation but provide few—if any—transactional semantics for operations on arbitrary sets of data items [11, 13, 15, 22, 26, 38, 44]. This results in incorrect behavior for use cases that require atomic visibility, including secondary indexing, foreign key constraint enforcement, and materialized view maintenance (Section 2). In contrast, many traditional transactional mechanisms correctly ensure atomicity of updates [8, 17, 43]. However, these algorithms—such as two-phase locking and variants of optimistic concurrency control—are often coordination-intensive, slow, and, under failure, unavailable in a distributed environment [5, 18, 28, 35]. This dichotomy between scalability and atomic visibility has been described as "a fact of life in the big cruel world of huge systems" [25]. The proliferation of non-transactional multi-item operations is symptomatic of a widespread "fear of synchronization" at scale [9].

Our contribution in this paper is to demonstrate that atomically visible transactions on partitioned databases are *not* at odds with scalability. Specifically, we provide high-performance implementations of a new, non-serializable isolation model called Read Atomic (RA) isolation. RA ensures that all or none of each transaction's updates are visible to others and that each transaction reads from an atomic snapshot of database state (Section 3)—this is useful in the applications we target. We subsequently develop three new, scalable algorithms for achieving RA isolation that we collectively title Read Atomic Multi-Partition (RAMP) transactions (Section 4). RAMP transactions guarantee scalability and outperform existing atomic algorithms because they satisfy two key scalability constraints. First, RAMP transactions guarantee *synchronization independence*: one client's transactions cannot cause another client's transactions to stall or fail. Second, RAMP transactions guarantee *partition independence*: clients never need to contact partitions that their transactions do not directly reference. Together, these properties ensure guaranteed completion, limited coordination across partitions, and horizontal scalability for multi-partition access.

RAMP transactions are scalable because they appropriately control the visibility of updates without inhibiting concurrency. Rather than force concurrent reads and writes to stall, RAMP transactions allow reads to "race" writes: RAMP transactions can autonomously detect the presence of non-atomic (partial) reads and, if necessary, repair them via a second round of communication with servers. To accomplish this, RAMP writers attach metadata to each write and use limited multi-versioning to prevent readers from stalling. The three algorithms we present offer a trade-off between the size of this metadata and performance. `RAMP-Small` transactions require constant space (a timestamp per write) and two round trip time delays (RTTs) for reads and writes. `RAMP-Fast` transactions require metadata size that is linear in the number of writes in the transaction but only require one RTT for reads in the common case and two in the worst case. `RAMP-Hybrid` transactions employ Bloom filters [10] to provide an intermediate solution. Traditional techniques like locking

[1] Our use of "atomic" (specifically, Read Atomic isolation) concerns all-or-nothing *visibility* of updates (i.e., the ACID isolation effects of ACID atomicity; Section 3). This differs from uses of "atomicity" to denote serializability [8] or linearizability [4].


*SIGMOD'14,* June 22–27, 2014, Snowbird, UT, USA.

ACM 978-1-4503-2376-5/14/06 ...$15.00.
http://dx.doi.org/10.1145/2588555.2588562.

couple atomic visibility and mutual exclusion; RAMP transactions provide the benefits of the former without incurring the scalability, availability, or latency penalties of the latter.

In addition to providing a theoretical analysis and proofs of correctness, we demonstrate that RAMP transactions deliver in practice. Our RAMP implementation achieves linear scalability to over 7 million operations per second on a 100 server cluster (at overhead below 5% for a workload of 95% reads). Moreover, across a range of workload configurations, RAMP transactions incur limited overhead compared to other techniques and achieve higher performance than existing approaches to atomic visibility (Section 5).

While the literature contains an abundance of isolation models [2, 5], we believe that the large number of modern applications requiring RA isolation and the excellent scalability of RAMP transactions justify the addition of yet another model. RA isolation is too weak for some applications, but, for the many that it can serve, RAMP transactions offer substantial benefits.

# 2. OVERVIEW AND MOTIVATION

In this paper, we consider the problem of making transactional updates atomically visible to readers—a requirement that, as we outline in this section, is found in several prominent use cases today. The basic property we provide is fairly simple: either all or none of each transaction's updates should be visible to other transactions. For example, if a transaction $T_1$ writes $x = 1$ and $y = 1$, then another transaction $T_2$ should not read $x = 1$ and $y = null$. Instead, $T_2$ should either read $x = 1$ and $y = 1$ or, possibly, $x = null$ and $y = null$. Informally, each transaction reads from an unchanging snapshot of database state that is aligned along transactional boundaries. We call this property *atomic visibility* and formalize it via the Read Atomic isolation guarantee in Section 3.

The classic strategy for providing atomic visibility is to ensure mutual exclusion between readers and writers. For example, if a transaction like $T_1$ above wants to update data items $x$ and $y$, it can acquire exclusive locks for each of $x$ and $y$, update both items, then release the locks. No other transactions will observe partial updates to $x$ and $y$, ensuring atomic visibility. However, this solution has a drawback: while one transaction holds exclusive locks on $x$ and $y$, no other transactions can access $x$ and $y$ for either reads or writes. By using mutual exclusion to enforce the atomic visibility of updates, we have also limited concurrency. In our example, if $x$ and $y$ are located on different servers, concurrent readers and writers will be unable to perform useful work during communication delays. These communication delays form an upper bound on throughput: effectively, $\frac{1}{\text{message delay}}$ operations per second.

To avoid this upper bound, we separate the problem of providing atomic visibility from the problem of maintaining mutual exclusion. By achieving the former but avoiding the latter, the algorithms we develop in this paper are not subject to the scalability penalties of many prior approaches. To ensure that all servers successfully execute a transaction (or that none do), our algorithms employ an atomic commitment protocol (ACP). When coupled with a blocking concurrency control mechanism like locking, ACPs are harmful to scalability and availability: arbitrary failures can (provably) cause any ACP implementation to stall [8]. (Optimistic concurrency control mechanisms can similarly block during validation.) We instead use ACPs with non-blocking concurrency control mechanisms; this means that individual transactions can stall due to failures or communication delays without forcing other transactions to stall. In a departure from traditional concurrency control, we allow multiple ACP rounds to proceed in parallel over the same data.

The end result—our RAMP transactions—provide excellent scalability and performance under contention (e.g., in the event of write hotspots) and are robust to partial failure. RAMP transactions' non-blocking behavior means that they cannot provide certain guarantees like preventing concurrent updates. However, applications that can use Read Atomic isolation will benefit from our algorithms. The remainder of this section identifies several relevant use cases from industry that require atomic visibility for correctness.

## 2.1 Read Atomic Isolation in the Wild

As a simple example, consider a social networking application: if two users, Sam and Mary, become "friends" (a bi-directional relationship), other users should never see that Sam is a friend of Mary but Mary is not a friend of Sam: either both relationships should be visible, or neither should be. A transaction under Read Atomic isolation would correctly enforce this behavior, and we can further classify three general use cases for Read Atomic isolation:

**1.) Foreign key constraints.** Many database schemas contain information about relationships between records in the form of foreign key constraints. For example, Facebook's TAO [11], LinkedIn's Espresso [38], and Yahoo! PNUTS [15] store information about business entities such as users, photos, and status updates as well as relationships between them (e.g., the friend relationships above). Their data models often represent bi-directional edges as two distinct uni-directional relationships. For example, in TAO, a user performing a "like" action on a Facebook page produces updates to both the `LIKES` and `LIKED_BY` associations [11]. PNUTS's authors describe an identical scenario [15]. These applications require foreign key maintenance and often, due to their unidirectional relationships, multi-entity update and access. Violations of atomic visibility surface as broken bi-directional relationships (as with Sam and Mary above) and dangling or incorrect references (e.g., Frank is an `employee` of `department.id=5`, but no such `department` exists in the `department` table).

With RAMP transactions, when inserting new entities, applications can bundle relevant entities from each side of a foreign key constraint into a transaction. When deleting associations, users can "tombstone" the opposite end of the association (i.e., delete any entries with associations via a special record that signifies deletion) [45] to avoid dangling pointers.

**2.) Secondary indexing.** Data is typically partitioned across servers according to a primary key (e.g., user ID). This allows fast location and retrieval of data via primary key lookups but makes access by secondary attributes (e.g., birth date) challenging. There are two dominant strategies for distributed secondary indexing. First, the *local secondary index* approach co-locates secondary indexes and primary data, so each server contains a secondary index that only references (and indexes) data stored on its server [7,38]. This allows easy, single-server updates but requires contacting every partition for secondary attribute lookups (write-one, read-all), compromising scalability for read-heavy workloads [11, 17, 38]. Alternatively, the *global secondary index* approach locates secondary indexes (which may be partitioned, but by a secondary attribute) separately from primary data [7, 15]. This alternative allows fast secondary lookups (read-one) but requires multi-partition update (at least write-two).

Real-world services employ either local secondary indexing (e.g., Espresso [38], Cassandra, and Google Megastore's local indexes [7]) or non-atomic (incorrect) global secondary indexing (e.g., Espresso and Megastore's global indexes, Yahoo! PNUTS's proposed secondary indexes [15]). The former is non-scalable but correct, while the latter is scalable but incorrect. For example, in a database partitioned by `id` with an incorrectly-maintained global secondary index on `salary`, the query `'SELECT id, salary WHERE salary > 60,000'` might return records with `salary` less than $60,000 and omit some records with `salary` greater than $60,000.

With RAMP transactions, the secondary index entry for a given attribute can be updated atomically with base data. For example, if a secondary index is stored as a mapping from secondary attribute values to sets of item-versions matching the secondary attribute (e.g., the secondary index entry for users with blue hair would contain a list of user IDs and last-modified timestamps corresponding to all of the users with attribute `hair-color=blue`), then insertions of new primary data require additions to the corresponding index entry, deletions require removals, and updates require a "tombstone" deletion from one entry and an insertion into another.

**3.) Materialized view maintenance.** Many applications precompute (i.e., materialize) queries over data, as in Twitter's Rainbird service [44], Google's Percolator [36], and LinkedIn's Espresso systems [38]. As a simple example, Espresso stores a mailbox of messages for each user along with statistics about the mailbox messages: for Espresso's read-mostly workload, it is more efficient to maintain (i.e., pre-materialize) a count of unread messages rather than scan all messages every time a user accesses her mailbox [38]. In this case, any unread message indicators should remain in sync with the messages in the mailbox. However, atomicity violations will allow materialized views to diverge from the base data (e.g., Susan's mailbox displays a notification that she has `unread` messages but all 63,201 messages in her `inbox` are marked as `read`).

With RAMP transactions, base data and views can be updated atomically. The physical maintenance of a view depends on its specification [14, 27], but RAMP transactions provide appropriate concurrency control primitives for ensuring that changes are delivered to the materialized view partition. For select-project views, a simple solution is to treat the view as a separate table and perform maintenance as needed: new rows can be inserted/deleted according to the specification, and, if necessary, the view can be (re-)computed on demand (i.e., lazy view maintenance [46]). For more complex views, such as counters, users can execute RAMP transactions over specialized data structures such as the CRDT G-Counter [40].

**In brief: Status Quo.** Despite application requirements for Read Atomic isolation, few large-scale production systems provide it. For example, the authors of Tao, Espresso, and PNUTS describe several classes of atomicity anomalies exposed by their systems, ranging from dangling pointers to the exposure of intermediate states and incorrect secondary index lookups, often highlighting these cases as areas for future research and design [11, 15, 38]. These systems are not exceptions: data stores like Bigtable [13], Dynamo [22], and many popular "NoSQL" [34] and even some "NewSQL" [5] stores do not provide transactional guarantees for multi-item operations.

The designers of these Internet-scale, real-world systems have made a conscious decision to provide scalability at the expense of multi-partition transactional semantics. Our goal with RAMP transactions is to preserve this scalability but deliver correct, atomically visible behavior for the use cases we have described.

# 3. SEMANTICS AND SYSTEM MODEL

In this section, we formalize Read Atomic isolation and, to capture scalability, formulate a pair of strict scalability criteria: synchronization and partition independence. Readers more interested in RAMP algorithms may wish to proceed to Section 4.

## 3.1 RA Isolation: Formal Specification

To formalize RA isolation, as is standard [2], we consider ordered sequences of reads and writes to arbitrary sets of items, or transactions. We call the set of items a transaction reads from and writes to its *read set* and *write set*. Each write creates a *version* of an item and we identify versions of items by a unique *timestamp* taken from a totally ordered set (e.g., rational numbers). Timestamps induce a total order on versions of each item (and a partial order *across* versions of different items). We denote version $i$ of item $x$ as $x_i$.

A transaction $T_j$ exhibits *fractured reads* if transaction $T_i$ writes versions $x_m$ and $y_n$ (in any order, with $x$ possibly but not necessarily equal to $y$), $T_j$ reads version $x_m$ and version $y_k$, and $k < n$.

A system provides *Read Atomic* isolation (RA) if it prevents fractured reads anomalies and also prevents transactions from reading uncommitted, aborted, or intermediate data. Thus, RA provides transactions with a "snapshot" view of the database that respects transaction boundaries (see the Appendix for more details, including a discussion of transitivity). RA is simply a restriction on write *visibility*—if the ACID "Atomicity" property requires that all or none of a transaction's updates are performed, RA requires that all or none of a transaction's updates are made visible to other transactions.

## 3.2 RA Implications and Limitations

As outlined in Section 2.1, RA isolation matches many of our use cases. However, RA is *not* sufficient for all applications. RA does not prevent concurrent updates or provide serial access to data items. For example, RA is an incorrect choice for an application that wishes to maintain positive bank account balances in the event of withdrawals. RA is a better fit for our "friend" operation because the operation is write-only and correct execution (i.e., inserting both records) is not conditional on concurrent updates.

From a programmer's perspective, we have found RA isolation to be most easily understandable (at least initially) with read-only and write-only transactions; after all, because RA allows concurrent writes, any values that are read might be changed at any time. However, read-write transactions are indeed well defined under RA.

## 3.3 System Model and Scalability

We consider databases that are partitioned, with the set of items in the database spread over multiple servers. Each item has a single logical copy, stored on a server—called the item's *partition*—whose identity can be calculated using the item. Clients forward operations on each item to the item's partition, where they are executed. Transaction execution terminates in *commit*, signaling success, or *abort*, signaling failure. In our examples, all data items have the null value ($\perp$) at database initialization. We do not model replication of data items within a partition; this can happen at a lower level of the system than our discussion (see Section 4.6) as long as operations on each item are linearizable [4].

**Scalability criteria.** As we hinted in Section 1, large-scale deployments often eschew transactional functionality on the premise that it would be too expensive or unstable in the presence of failure and degraded operating modes [9, 11, 13, 15, 22, 25, 26, 38, 44]. Our goal in this paper is to provide robust and scalable transactional functionality, and, so we first define criteria for "scalability":

*Synchronization independence* ensures that one client's transactions cannot cause another client's to block and that, if a client can contact the partition responsible for each item in its transaction, the transaction will eventually commit (or abort of its own volition). This prevents one transaction from causing another to abort—which is particularly important in the presence of partial failures—and guarantees that each client is able to make useful progress. In the absence of failures, this maximizes useful concurrency. In the distributed systems literature, synchronization independence for replicated transactions is called *transactional availability* [5]. Note that "strong" isolation models like serializability and Snapshot Isolation violate synchronization independence and limit scalability.

While many applications can limit their data accesses to a single partition via explicit data modeling [7,19,25,38] or planning [18,35], this is not always possible. In the case of secondary indexing, there is a tangible cost associated with requiring single-partition updates (scatter-gather reads), while, in social networks like Facebook and large-scale hierarchical access patterns as in Rainbird, perfect partitioning of data accesses is near-impossible. Accordingly:

*Partition independence* ensures that, in order to execute a transaction, a client never has to contact partitions that its transaction does not access. Thus, a partition failure only affects transactions that access items contained on the partition. This also reduces load on servers not directly involved in a transaction's execution. In the distributed systems literature, partition independence for replicated data is called *replica availability* [5] or *genuine partial replication* [39].

In addition to the above requirements, we limit the *metadata overhead* of algorithms. There are many potential solutions for providing atomic visibility that rely on storing prohibitive amounts of state. As a straw-man solution, each transaction could send copies of all of its writes to every partition it accesses so that readers observe all of its writes by reading a single item. This provides RA isolation but requires considerable storage. Other solutions may require extra data storage proportional to the number of servers in the cluster or, worse, the database size (Section 6). We will attempt to minimize this *metadata*—that is, data that the transaction did not itself write but which is required for correct execution. In our algorithms, we will specifically provide constant-factor metadata overheads (RAMP-S, RAMP-H) or else overhead linear in transaction size (but independent of data size; RAMP-F).

# 4. RAMP TRANSACTION ALGORITHMS

Given specifications for RA isolation and scalability, we present algorithms for achieving both. For ease of understanding, we first focus on providing read-only and write-only transactions with a "last writer wins" overwrite policy, then subsequently discuss how to perform read/write transactions. Our focus in this section is on intuition and understanding; we defer all correctness and scalability proofs to the Appendix, providing salient details inline.

At a high level, RAMP transactions allow reads and writes to proceed concurrently. This provides excellent performance but, in turn, introduces a race condition: one transaction might only read a subset of another transaction's writes, violating RA (i.e., fractured reads might occur). Instead of preventing this race (hampering scalability), RAMP readers autonomously detect the race (using metadata attached to each data item) and fetch any missing, in-flight writes from their respective partitions. To make sure that readers never have to block for writes to arrive at a partition, writers use a two-phase (atomic commitment) protocol that ensures that once a write is visible to readers on one partition, any other writes in the transaction are present on and, if appropriately identified by version, readable from their respective partitions.

In this section, we present three algorithms that provide a trade-off between the amount of metadata required and the expected number of extra reads to fetch missing writes. As discussed in Section 2, if techniques like distributed locking couple mutual exclusion with atomic visibility of writes, RAMP transactions correctly control visibility but allow concurrent and scalable execution.

## 4.1 RAMP-Fast

To begin, we present a RAMP algorithm that, in the race-free case, requires one RTT for reads and two RTTs for writes, called RAMP-Fast (abbreviated RAMP-F; Algorithm 1). RAMP-F stores metadata in the form of write sets (overhead linear in transaction size).

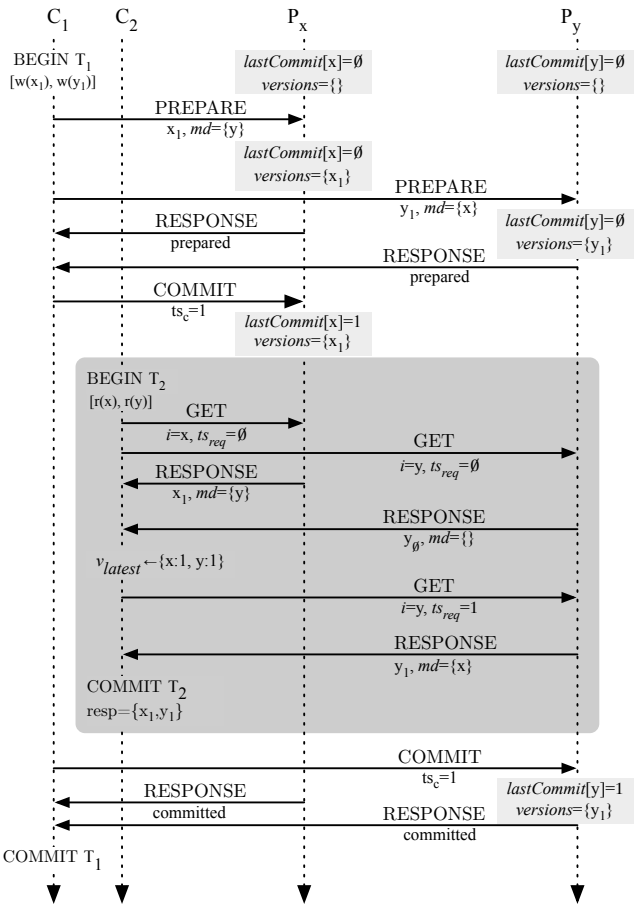


**Figure 1: Space-time diagram for RAMP-F execution for two transactions $T_1$ and $T_2$ performed by clients $C_1$ and $C_2$ on partitions $P_x$ and $P_y$. Because $T_1$ overlaps with $T_2$, $T_2$ must perform a second round of reads to repair the fractured read between $x$ and $y$. $T_1$'s writes are assigned timestamp 1. Lightly-shaded boxes represent current partition state (*lastCommit* and *versions*), while the single darkly-shaded box encapsulates all messages exchanged during $C_2$'s execution of transaction $T_2$.**

**Overview.** Each write in RAMP-F (lines 14–21) contains a timestamp (line 15) that uniquely identifies the writing transaction as well as a set of items written in the transaction (line 16). For now, combining a unique client ID and client-local sequence number is sufficient for timestamp generation (see also Section 4.5).

RAMP-F write transactions proceed in two phases: a first round of communication places each timestamped write on its respective partition. In this PREPARE phase, each partition adds the write to its local database (*versions*, lines 1, 17–19). A second round of communication marks versions as committed. In this COMMIT phase, each partition updates an index containing the highest-timestamped committed version of each item (*lastCommit*, lines 2, 20–21).

RAMP-F read transactions begin by first fetching the last (highest-timestamped) committed version for each item from its respective partition (lines 23–30). Using the results from this first round of reads, each reader can calculate whether it is "missing" any versions (that is, versions that were prepared but not yet committed on their partitions). Combining the timestamp and set of items from each version read (i.e., its metadata) produces a mapping from items to timestamps that represent the highest-timestamped write for each transaction that appears in this first-round read set (lines 26–29). If

the reader has read a version of an item that has a lower timestamp than indicated in the mapping for that item, the reader issues a second read to fetch the missing version (by timestamp) from its partition (lines 30–32). Once all missing versions are fetched (which can be done in parallel), the client can return the resulting set of versions—the first-round reads, with any missing versions replaced by the optional, second round of reads.

**By example.** Consider the RAMP-F execution depicted in Figure 1. $T_1$ writes to both $x$ and $y$, performing the two-round write protocol on two partitions, $P_x$ and $P_y$. However, $T_2$ reads from $x$ and $y$ while $T_1$ is concurrently writing. Specifically, $T_2$ reads from $P_x$ *after* $P_x$ has committed $T_1$'s write to $x$, but $T_2$ reads from $P_y$ *before* $P_y$ has committed $T_1$'s write to $y$. Therefore, $T_2$'s first-round reads return $x = x_1$ and $y = \bot$, and returning this set of reads would violate RA. Using the metadata attached to its first-round reads, $T_2$ determines that it is missing $y_1$ (since $v_{latest}[y] = 1$ and $1 > \bot$) and so $T_2$ subsequently issues a second read from $P_y$ to fetch $y_1$ by version. After completing its second-round read, $T_2$ can safely return its result set. $T_1$'s progress is unaffected by $T_2$, and $T_1$ subsequently completes by committing $y_1$ on $P_y$.

**Why it works.** RAMP-F writers use metadata as a record of intent: a reader can detect if it has raced with an in-progress commit round and use the metadata stored by the writer to fetch the missing data. Accordingly, RAMP-F readers only issue a second round of reads in the event that they read from a partially-committed write transaction (where some but not all partitions have committed a write). In this event, readers will fetch the appropriate writes from the not-yet-committed partitions. Most importantly, RAMP-F readers never have to stall waiting for a write that has not yet arrived at a partition: the two-round RAMP-F write protocol guarantees that, if a partition commits a write, all of the corresponding writes in the transaction are present on their respective partitions (though possibly not committed locally). As long as a reader can identify the corresponding version by timestamp, the reader can fetch the version from the respective partition's set of pending writes without waiting. To enable this, RAMP-F writes contain metadata linear in the size of the writing transaction's write set (plus a timestamp per write).

RAMP-F requires 2 RTTs for writes: one for PREPARE and one for COMMIT. For reads, RAMP-F requires one RTT in the absence of concurrent writes and two RTTs otherwise.

RAMP timestamps are only used to identify specific versions and in ordering concurrent writes to the same item; RAMP-F transactions do not require a "global" timestamp authority. For example, if $lastCommit[k] = 2$, there is no requirement that a transaction with timestamp 1 has committed or even that such a transaction exists.

## 4.2 RAMP-Small: Trading Metadata for RTTs

While RAMP-F requires linearly-sized metadata but provides best-case one RTT for reads, RAMP-Small (RAMP-S) uses constant-size metadata but always requires two RTT for reads (Algorithm 2). RAMP-S and RAMP-F writes are identical, but, instead of attaching the entire write set to each write, RAMP-S writers only store the transaction timestamp (line 7). Unlike RAMP-F, RAMP-S readers issue a first round of reads to fetch the highest committed timestamp for each item from its respective partition (lines 3, 9–11). Once RAMP-S readers have recieved the highest committed timestamp for each item, the readers send the entire set of timestamps they received to the partitions in a second round of communication (lines 13–14). For each item in the read request, RAMP-S servers return the highest-timestamped version of the item that also appears in the supplied set of timestamps (lines 5–6). Readers subsequently return the results from the mandatory second round of requests.

**Algorithm 1** RAMP-Fast

***Server-side Data Structures***

1: *versions*: set of versions $\langle item, value$, timestamp $ts_v$, metadata $md\rangle$
2: $latestCommit[i]$: last committed timestamp for item $i$

***Server-side Methods***

3: **procedure** PREPARE($v$ : version)
4: $versions$.add($v$)
5: **return**

6: **procedure** COMMIT($ts_c$ : timestamp)
7: $I_{ts} \leftarrow \{w.item \mid w \in versions \wedge w.ts_v = ts_c\}$
8: $\forall i \in I_{ts}$, $latestCommit[i] \leftarrow \max(latestCommit[i], ts_c)$

9: **procedure** GET($i$ : item, $ts_{req}$ : timestamp)
10: **if** $ts_{req} = \emptyset$ **then**
11: **return** $v \in versions : v.item = i \wedge v.ts_v = latestCommit[item]$
12: **else**
13: **return** $v \in versions : v.item = i \wedge v.ts_v = ts_{req}$

***Client-side Methods***

14: **procedure** PUT_ALL($W$ : set of $\langle item, value\rangle$)
15: $ts_{tx} \leftarrow$ generate new timestamp
16: $I_{tx} \leftarrow$ set of items in $W$
17: **parallel-for** $\langle i, v\rangle \in W$
18: $v \leftarrow \langle item = i, value = v, ts_v = ts_{tx}, md = (I_{tx} - \{i\})\rangle$
19: invoke PREPARE($v$) on respective server (i.e., partition)
20: **parallel-for** server $s$ : $s$ contains an item in $W$
21: invoke COMMIT($ts_{tx}$) on $s$

22: **procedure** GET_ALL($I$ : set of items)
23: $ret \leftarrow \{\}$
24: **parallel-for** $i \in I$
25: $ret[i] \leftarrow$ GET($i, \emptyset$)
26: $v_{latest} \leftarrow \{\}$ (default value: $-1$)
27: **for** response $r \in ret$ **do**
28: **for** $i_{tx} \in r.md$ **do**
29: $v_{latest}[i_{tx}] \leftarrow \max(v_{latest}[i_{tx}], r.ts_v)$
30: **parallel-for** item $i \in I$
31: **if** $v_{latest}[i] > ret[i].ts_v$ **then**
32: $ret[i] \leftarrow$ GET($i, v_{latest}[i]$)
33: **return** $ret$

**By example.** In Figure 1, under RAMP-S, $P_x$ and $P_y$ would respectively return the sets $\{1\}$ and $\{\bot\}$ in response to $T_2$'s first round of reads. $T_2$ would subsequently send the set $\{1, \bot\}$ to both $P_x$ and $P_y$, which would return $x_1$ and $y_1$. (Including $\bot$ in the second-round request is unnecessary, but we leave it in for ease of understanding.)

**Why it works.** In RAMP-S, if a transaction has committed on some but not all partitions, the transaction timestamp will be returned in the first round of any concurrent read transaction accessing the committed partitions' items. In the (required) second round of read requests, any prepared-but-not-committed partitions will find the committed timestamp in the reader-provided set and return the appropriate version. In contrast with RAMP-F, where readers explicitly provide partitions with a specific version to return in the (optional) second round, RAMP-S readers defer the decision of which version to return to the partition, which uses the reader-provided set to decide. This saves metadata but increases RTTs, and the size of the parameters of each second-round GET request is (worst-case) linear in the read set size. Unlike RAMP-F, there is no requirement to return the value of the last committed version in the first round (returning the version, $lastCommit[k]$, suffices in line 3).

## 4.3 RAMP-Hybrid: An Intermediate Solution

RAMP-Hybrid (RAMP-H; Algorithm 3) strikes a compromise between RAMP-F and RAMP-S. RAMP-H and RAMP-S write protocols are identical, but, instead of storing the entire write set (as in RAMP-F),

**Algorithm 2** RAMP-Small

***Server-side Data Structures***
same as RAMP-F (Algorithm 1)

***Server-side Methods***
PREPARE, COMMIT same as in RAMP-F

```
1:  procedure GET(i : item, ts_set : set of timestamps)
2:      if ts_set = ∅ then
3:          return v ∈ versions : v.item = i ∧ v.ts_v = latestCommit[k]
4:      else
5:          ts_match = {t | t ∈ ts_set ∧ ∃v ∈ versions : v.item = i ∧ v.t_v = t}
6:          return v ∈ versions : v.item = i ∧ v.ts_v = max(ts_match)
```

***Client-side Methods***

```
7:  procedure PUT_ALL(W : set of ⟨item, value⟩)
        same as RAMP-F PUT_ALL but do not instantiate md on line 18
8:  procedure GET_ALL(I : set of items)
9:      ts_set ← {}
10:     parallel-for i ∈ I
11:         ts_set.add(GET(i, ∅).ts_v)
12:     ret ← {}
13:     parallel-for item i ∈ I
14:         ret[i] ← GET(i, ts_set)
15:     return ret
```

RAMP-H writers store a Bloom filter [10] representing the transaction write set (line 1). RAMP-H readers proceed as in RAMP-F, with a first round of communication to fetch the last-committed version of each item from its partition (lines 3–5). Given this set of versions, RAMP-H readers subsequently compute a list of *potentially* higher-timestamped writes for each item (lines 7–10). Any potentially missing versions are fetched in a second round of reads (lines 12).

**By example.** In Figure 1, under RAMP-H, $x_1$ would contain a Bloom filter with positives for $x$ and $y$ and $y_\perp$ would contain an empty Bloom filter. $T_2$ would check for the presence of $y$ in $x_1$'s Bloom filter (since $x_1$'s version is 1 and $1 > \perp$) and, finding a match, conclude that it is potentially missing a write ($y_1$). $T_2$ would subsequently fetch $y_1$ from $P_y$.

**Why it works.** RAMP-H is effectively a hybrid between RAMP-F and RAMP-S. If the Bloom filter has no false positives, RAMP-H reads behave like RAMP-F reads. If the Bloom filter has all false positives, RAMP-H reads behave like RAMP-S reads. Accordingly, the number of (unnecessary) second-round reads (i.e., which would not be performed by RAMP-F) is controlled by the Bloom filter false positive rate, which is in turn (in expectation) proportional to the size of the Bloom filter. Any second-round GET requests are accompanied by a set of timestamps that is also proportional in size to the false positive rate. Therefore, RAMP-H exposes a trade-off between metadata size and expected performance. To understand why RAMP-H is safe, we simply have to show that any false positives (second-round reads) will not compromise the integrity of the result set; with unique timestamps, any reads due to false positives will return null.

## 4.4 Summary of Basic Algorithms

The RAMP algorithms allow readers to safely race writers without requiring either to stall. The metadata attached to each write allows readers in all three algorithms to safely handle concurrent and/or partial writes and in turn allows a trade-off between metadata size and performance (Table 1): RAMP-F is optimized for fast reads, RAMP-S is optimized for small metadata, and RAMP-H is, as the name suggests, a middle ground. RAMP-F requires metadata linear in transaction size, while RAMP-S and RAMP-H require constant metadata. However, RAMP-S and RAMP-H require more RTTs for reads compared to RAMP-F when there is no race between readers and writers.

**Algorithm 3** RAMP-Hybrid

***Server-side Data Structures***
Same as in RAMP-F (Algorithm 1)

***Server-side Methods***
PREPARE, COMMIT same as in RAMP-F
GET same as in RAMP-S

***Client-side Methods***

```
1:  procedure PUT_ALL(W : set of ⟨item, value⟩)
        same as RAMP-F PUT_ALL but instantiate md on line 18
        with Bloom filter containing I_tx
2:  procedure GET_ALL(I : set of items)
3:      ret ← {}
4:      parallel-for i ∈ I
5:          ret[i] ← GET(i, ∅)
6:      v_fetch ← {}
7:      for version v ∈ ret do
8:          for version v' ∈ ret : v' ≠ v do
9:              if v.ts_v > v'.ts_v ∧ v.md.lookup(v'.item) → True then
10:                 v_fetch[v'.item].add(v.ts_v)
11:     parallel-for item i ∈ v_fetch
12:         ret[i] ← GET(k, v_fetch[i]) if GET(k, v_fetch[i]) ≠ ⊥
13:     return ret
```

| Algorithm | RTTs/transaction | | | Metadata (+stamp) | |
|---|---|---|---|---|---|
| | W | R (stable) | R ($O$) | Stored | Per-Request |
| RAMP-F | 2 | 1 | 2 | txn items | - |
| RAMP-S | 2 | 2 | 2 | - | stamp/item |
| RAMP-H | 2 | $1+\varepsilon$ | 2 | Bloom filter | stamp/item |

**Table 1: Comparison of basic algorithms: RTTs required for writes (W), reads (R) without concurrent writes and in the worst case ($O$), stored metadata and metadata attached to read requests (in addition to a timestamp for each).**

When reads and writes race, in the worst case, all algorithms require two RTTs for reads. Writes always require two RTTs to prevent readers from stalling due to missing, unprepared writes.

RAMP algorithms are scalable because clients only contact partitions relative to their transactions (partition independence), and clients cannot stall one another (synchronization independence). More specifically, readers do not interfere with other readers, writers do not interfere with other writers, and readers and writers can proceed concurrently. When a reader races a writer to the same items, the writer's new versions will only become visible to the reader (i.e., be committed) once it is guaranteed that the reader will be able to fetch all of them (possibly via a second round of communication). A reader will *never* have to stall waiting for writes to arrive at a partition (for details, see Invariant 1 in the Appendix).

## 4.5 Additional Details

In this section, we discuss relevant implementation details.

**Multi-versioning and garbage collection.** RAMP transactions rely on multi-versioning to allow readers to access versions that have not yet committed and/or have been overwritten. In our initial presentation, we have used a completely multi-versioned storage engine; in practice, multi-versioning can be implemented by using a single-versioned storage engine for retaining the last committed version of each item and using a "look-aside" store for access to both prepared-but-not-yet-committed writes and (temporarily) any overwritten versions. The look-aside store should make prepared versions durable but can—at the risk of aborting transactions in the event of a server failure—simply store any overwritten versions in

memory. Thus, with some work, RAMP algorithms are portable to legacy, non-multi-versioned storage systems.

In both architectures, each partition's data will grow without bound if old versions are not removed. If a committed version of an item is not the highest-timestamped committed version (i.e., a committed version $v$ of item $k$ where $v < lastCommit[k]$), it can be safely discarded (i.e., garbage collected, or GCed) as long as no readers will attempt to access it in the future (via second-round GET requests). It is easiest to simply limit the running time of read transactions and GC overwritten versions after a fixed amount of real time has elapsed. Any read transactions that take longer than this GC window can be restarted [32, 33]. Therefore, the maximum number of versions retained for each item is bounded by the item's update rate, and servers can reject any client GET requests for versions that have been GCed (and the read transaction can be restarted). As a more principled solution, partitions can also gossip the timestamps of items that have been overwritten and have not been returned in the first round of any ongoing read transactions.

**Read-write transactions.** Until now, we have focused on read-only and write-only transactions. However, we can extend our algorithms to provide read-write transactions. If transactions pre-declare the data items they wish to read, then the client can execute a GET_ALL transaction at the start of transaction execution to pre-fetch all items; subsequent accesses to those items can be served from this pre-fetched set. Clients can buffer any writes and, upon transaction commit, send all new versions to servers (in parallel) via a PUT_ALL request. As in Section 3, this may result in anomalies due to concurrent update but does not violate RA isolation. Given the benefits of pre-declared read/write sets [18, 35, 43] and write buffering [17, 41], we believe this is a reasonable strategy. For secondary index lookups, clients can first look up secondary index entries then subsequently (within the same transaction) read primary data (specifying versions from index entries as appropriate).

**Timestamps.** Timestamps should be unique across transactions, and, for "session" consistency (Appendix), increase on a per-client basis. Given unique client IDs, a client ID and sequence number form unique transaction timestamps without coordination. Without unique client IDs, servers can assign unique timestamps with high probability using UUIDs and by hashing transaction contents.

**Overwrites.** In our algorithms, we have depicted a policy in which versions are overwritten according to a highest-timestamp-wins policy. In practice, and, for commutative updates, users may wish to employ a different policy upon COMMIT: for example, perform set union. In this case, $lastCommit[k]$ contains an abstract data type (e.g., set of versions) that can be updated with a *merge* operation [22, 42] (instead of $updateIfGreater$) upon commit. This treats each committed record as a set of versions, requiring additional metadata (that can be GCed as in Section 4.7).

## 4.6 Distribution and Fault Tolerance

RAMP transactions operate in a distributed setting, which poses challenges due to latency, partial failure, and network partitions. Synchronization independence ensures that failed clients do not cause other clients to fail, while partition independence ensures that clients only have to contact partitions for items in their transactions. This provides fault tolerance and availability as long as clients can access relevant partitions, but here we further elucidate RAMP interactions with replication and stalled operations.

**Replication.** A variety of mechanisms including traditional database master-slave replication with failover, quorum-based protocols, and state machine replication and can ensure availability of individual partitions in the event of individual server failure [8]. To control durability, clients can wait until the effects of their operations (e.g., modifications to *versions* and *lastCommit*) are persisted locally on their respective partitions and/or to multiple physical servers before returning from PUT_ALL calls (either via master-to-slave replication or via quorum replication and by performing two-phase commit across multiple active servers). Notably, because RAMP transactions can safely overlap in time, replicas can process different transactions' PREPARE and COMMIT requests in parallel.

**Stalled Operations.** RAMP writes use a two-phase atomic commitment protocol that ensures readers never block waiting for writes to arrive. As discussed in Section 2, every ACP may block during failures [8]. However, due to synchronization independence, a blocked transaction (due to failed clients, failed servers, or network partitions) cannot cause other transactions to block. Blocked writes instead act as "resource leaks" on partitions: partitions will retain prepared versions indefinitely unless action is taken.

To "free" these leaks, RAMP servers can use the Cooperative Termination Protocol (CTP) described in [8]. CTP can always complete the transaction except when every partition has performed PREPARE but no partition has performed COMMIT. In CTP, if a server $S_p$ has performed PREPARE for transaction $T$ but times out waiting for a COMMIT, $S_p$ can check the status of $T$ on any other partitions for items in $T$'s write set. If another server $S_c$ has received COMMIT for $T$, then $S_p$ can COMMIT $T$. If $S_a$, a server responsible for an item in $T$, has not received PREPARE for $T$, $S_a$ and $S_p$ can promise never to PREPARE or COMMIT $T$ in the future and $S_p$ can safely discard its versions. A client recovering from a failure can read from the servers to determine if they unblocked its write. Writes that block mid-COMMIT will also become visible on all partitions.

CTP (evaluated in Section 5) only runs when writes block (or time-outs fire) and runs *asynchronously* with respect to other operations. CTP requires that PREPARE messages contain a list of servers involved in the transaction (a subset of RAMP-F metadata but a superset of RAMP-H and RAMP-S) and that servers remember when they COMMIT and "abort" writes (e.g., in a log file). Compared to alternatives (e.g., replicating clients [24]), we have found CTP to be both lightweight and effective.

## 4.7 Further Optimizations

RAMP algorithms also allow several possible optimizations:

**Faster commit detection.** If a server returns a version in response to a GET request and the version's timestamp is greater than the highest committed version of that item (i.e., *lastCommit*), then transaction writing the version has committed on at least one partition. In this case, the server can mark the version as committed. This scenario will occur when all partitions have performed PREPARE and at least one server but not all partitions have performed COMMIT (as in CTP). This allows faster updates to *lastCommit* (and therefore fewer expected RAMP-F and RAMP-H RTTs).

**Metadata garbage collection.** Once all of transaction $T$'s writes are committed on each respective partition (i.e., are reflected in *lastCommit*), readers are guaranteed to read $T$'s writes (or later writes). Therefore, non-timestamp metadata for $T$'s writes stored in RAMP-F and RAMP-H (write sets and Bloom filters) can therefore be discarded. Detecting that all servers have performed COMMIT can be performed asynchronously via a third round of communication performed by either clients or servers.

**One-phase writes.** We have considered two-phase writes, but, if a user does not wish to read her writes (thereby sacrificing session guarantees outlined in the Appendix), the client can return after issuing its PREPARE round (without sacrificing durability). The client can subsequently execute the COMMIT phase asynchronously,

or, similar to optimizations presented in Paxos Commit [24], the servers can exchange PREPARE acknowledgements with one another and decide to COMMIT autonomously. This optimization is safe because multiple PREPARE phases can safely overlap.

# 5. EXPERIMENTAL EVALUATION

We proceed to experimentally demonstrate RAMP transaction scalability as compared to existing transactional and non-transactional mechanisms. RAMP-F, RAMP-H, and often RAMP-S outperform existing solutions across a range of workload conditions while exhibiting overheads typically within 8% and no more than 48% of peak throughput. As expected from our theoretical analysis, the performance of our RAMP algorithms does not degrade substantially under contention and scales linearly to over 7.1 million operations per second on 100 servers. These outcomes validate our choice to pursue synchronization- and partition-independent algorithms.

## 5.1 Experimental Setup

To demonstrate the effect of concurrency control on performance and scalability, we implemented several concurrency control algorithms in a partitioned, multi-versioned, main-memory database prototype. Our prototype is in Java and employs a custom RPC system with Kryo 2.20 for serialization. Servers are arranged as a distributed hash table with partition placement determined by random hashing. As in stores like Dynamo [22], clients can connect to any server to execute operations, which the server will perform on their behalf (i.e., each server acts as a client in our RAMP pseudocode). We implemented RAMP-F, RAMP-S, and RAMP-H and configure a wall-clock GC window of 5 seconds as described in Section 4.5. RAMP-H uses a 256-bit Bloom filter based on an implementation of MurmurHash2.0, with four hashes per entry; to demonstrate the effects of filter saturation, we do not modify these parameters in our experiments. Our prototype utilizes the "Faster commit detection" optimization from Section 4.5 but we chose not to employ the latter two optimizations in order to preserve session guarantees and because metadata overheads were generally minor.

**Algorithms for comparison.** As a baseline, we do not employ any concurrency control (denoted NWNR, for no write and no read locks); reads and writes take one RTT and are executed in parallel.

We also consider three lock-based mechanisms: long write locks and long read locks, providing Repeatable Read isolation (*PL-2.99*; denoted LWLR), long write locks with short read locks, providing Read Committed isolation (*PL-2L*; denoted LWSR; does not provide RA), and long write locks with no read locks, providing Read Uncommitted isolation [2] (LWNR; also does not provide RA). While only LWLR provides RA, LWSR and LWNR provide a useful basis for comparison, particularly in measuring concurrency-related locking overheads. To avoid deadlocks, the system lexicographically orders lock requests by item and performs them sequentially. When locks are not used (as for reads in LWNR and reads and writes for NWNR), the system parallelizes operations.

We also consider an algorithm where, for each transaction, designated "coordinator" servers enforce RA isolation—effectively, the Eiger system's 2PC-PCI mechanism [33] (denoted E-PCI; Section 6). Writes proceed via prepare and commit rounds, but any reads that arrive at a partition and overlap with a concurrent write to the same item must contact a (randomly chosen, per-write-transaction) "coordinator" partition to determine whether the coordinator's prepared writes have been committed. Writes require two RTTs, while reads require one RTT during quiescence and two RTTs in the presence of concurrent updates (to a variable number of coordinator partitions—linear in the number of concurrent writes to the item). Using a coordinator violates partition independence but not synchronization independence. We optimize 2PC-PCI reads by having clients determine a read timestamp for each transaction (eliminating an RTT) and do not include happens-before metadata.

This range of lock-based strategies (LWNR, LWSR, LWNR), recent comparable approach (E-PCI), and best-case (NWNR; no concurrency control) baseline provides a spectrum of strategies for comparison.

**Environment and benchmark.** We evaluate each algorithm using the YCSB benchmark [16] and deploy variably-sized sets of servers on public cloud infrastructure. We employ cr1.8xlarge instances on Amazon EC2 and, by default, deploy five partitions on five servers. We group sets of reads and sets of writes into read-only and write-only transactions (default size: 4 operations), and use the default YCSB workload (workloada, with Zipfian distributed item accesses) but with a 95% read and 5% write proportion, reflecting read-heavy applications (Section 2, [11, 33, 44]; e.g., Tao's 500 to 1 reads-to-writes [11, 33], Espresso's 1000 to 1 Mailbox application [38], and Spanner's 3396 to 1 advertising application [17]).

By default, we use 5000 concurrent clients split across 5 separate EC2 instances and, to fully expose our metadata overheads, use a value size of 1 byte per write. We found that lock-based algorithms were highly inefficient for YCSB's default 1K item database, so we increased the database size to 1M items by default. Each version contains a timestamp (64 bits), and, with YCSB keys (i.e., item IDs) of size 11 bytes and a transaction length $L$, RAMP-F requires $11L$ bytes of metadata per version, while RAMP-H requires 32 bytes. We successively vary several parameters, including number of clients, read proportion, transaction length, value size, database size, and number of servers and report the average of three sixty-second trials.

## 5.2 Experimental Results: Comparison

Our first set of experiments focuses on two metrics: performance compared to baseline and performance compared to existing techniques. The overhead of RAMP algorithms is typically less than 8% compared to baseline (NWNR) throughput, is sometimes zero, and is never greater than 50%. RAMP-F and RAMP-H always outperform the lock-based and E-PCI techniques, while RAMP-S outperforms lock-based techniques and often outperforms E-PCI. We proceed to demonstrate this behavior over a variety of conditions:

**Number of clients.** RAMP performance scales well with increased load and incurs little overhead (Figure 2). With few concurrent clients, there are few concurrent updates and therefore few second-round reads; performance for RAMP-F and RAMP-H is close to or even matches that of NWNR. At peak throughput (at 10,000 clients), RAMP-F and RAMP-H pay a throughput overhead of 4.2% compared to NWNR. RAMP-F and RAMP-H exhibit near-identical performance; the RAMP-H Bloom filter triggers few false positives (and therefore few extra RTTs compared to RAMP-F). RAMP-S incurs greater overhead and peaks at almost 60% of the throughput of NWNR. Its guaranteed two-round trip reads are expensive and it acts as an effective lower bound on RAMP-F and RAMP-H performance. In all configurations, the algorithms achieve low latency (RAMP-F, RAMP-H, NWNR less than 35ms on average and less than 10 ms at 5,000 clients; RAMP-S less than 53ms, 14.3 ms at 5,000 clients).

In comparison, the remaining algorithms perform less favorably. In contrast with the RAMP algorithms, E-PCI servers must check a coordinator server for each in-flight write transaction to determine whether to reveal writes to clients. For modest load, the overhead of these commit checks places E-PCI performance between that of RAMP-S and RAMP-H. However, the number of in-flight writes increases with load (and is worsened due to YCSB's Zipfian distributed accesses), increasing the number of E-PCI commit checks.

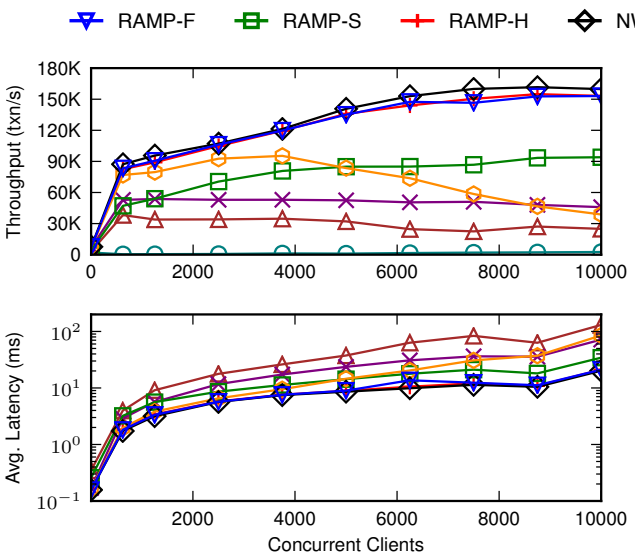


**Figure 2: Throughput and latency under varying client load. We omit latencies for LWLR, which peaked at over 1.5s.**

This in turn decreases throughput, and, with 10,000 concurrent clients, E-PCI performs so many commit checks per read (over 20% of reads trigger a commit check, and, on servers with hot items, each commit check requires indirected coordinator checks for an average of 9.84 transactions) that it underperforms the LWNR lock-based scheme. Meanwhile, multi-partition locking is expensive [35]: with 10,000 clients, the most efficient algorithm, LWNR, attains only 28.6% of the throughput of NWNR, while the least efficient, LWLR, attains only 1.6% (peaking at 3,412 transactions per second).

We subsequently varied several other workload parameters, which we briefly discuss below and plot in Figure 3:

**Read proportion.** Increased write activity leads to a greater number of races between reads and writes and therefore additional second-round RTTs for RAMP-F and RAMP-H reads. With all write transactions, all RAMP algorithms are equivalent (two RTT) and achieve approximately 65% of the throughput of NWNR. With all reads, RAMP-F, RAMP-S, NWNR, and E-PCI are identical, with a single RTT. Between these extremes, RAMP-F and RAMP-S scale near-linearly with the write proportion. In contrast, lock-based protocols fare poorly as contention increases, while E-PCI again incurs penalties due to commit checks.

**Transaction length.** Increased transaction lengths have variable impact on the relative performance of RAMP algorithms. Synchronization independence does not penalize long-running transactions, but, with longer transactions, metadata overheads increase. RAMP-F relative throughput decreases due to additional metadata (linear in transaction length) and RAMP-H relative performance also decreases as its Bloom filters saturate. (However, YCSB's Zipfian-distributed access patterns result in a non-linear relationship between length and throughput.) As discussed above, we explicitly decided not to tune RAMP-H Bloom filter size but believe a logarithmic increase in filter size could improve RAMP-H performance for large transaction lengths (e.g., 1024 bit filters should lower the false positive rate for transactions of length 256 from over 92% to slightly over 2%).

**Value size.** Value size similarly does not seriously impact relative throughput. At a value size of 1B, RAMP-F is within 2.3% of NWNR. However, at a value size of 100KB, RAMP-F performance nearly matches that of NWNR: the overhead due to metadata decreases, and write request rates slow, decreasing concurrent writes (and subsequently second-round RTTs). Nonetheless, absolute throughput drops by a factor of 24 as value sizes moves from 1B to 100KB.

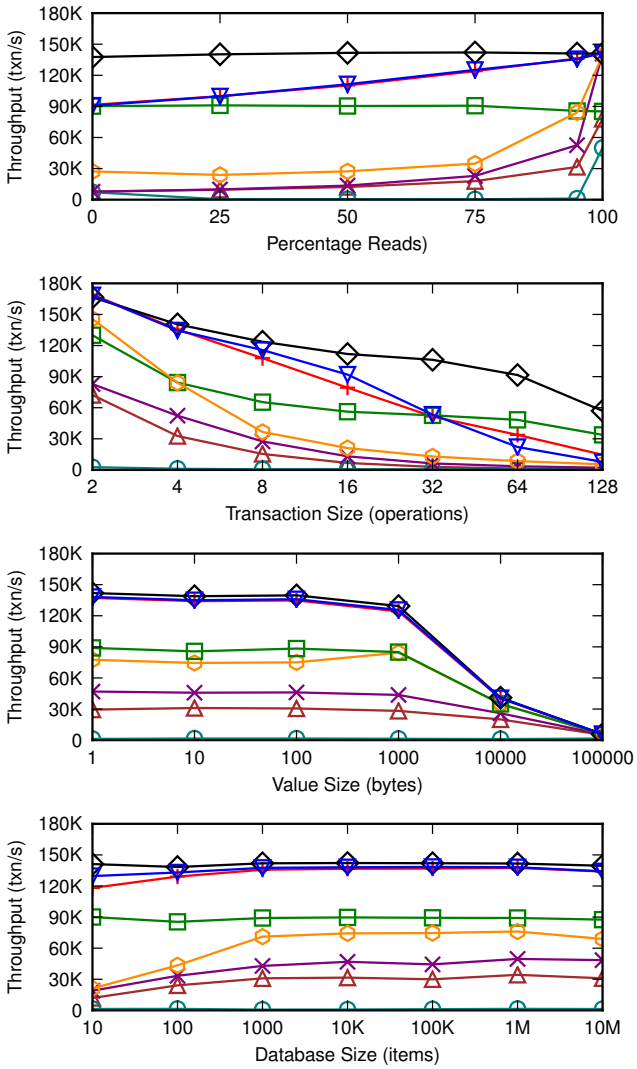


**Figure 3: Algorithm performance across varying workload conditions. RAMP-F and RAMP-H exhibit similar performance to NWNR baseline, while RAMP-S's 2 RTT reads incur a greater performance penalty across almost all configurations. RAMP transactions consistently outperform RA isolated alternatives.**

**Database size.** RAMP algorithms are robust to high contention for a small set of items: with only 1000 items in the database, RAMP-F achieves throughput within 3.1% of NWNR. RAMP algorithms are largely agnostic to read/write contention, although, with fewer items in the database, the probability of races between readers and in-progress writers increases, resulting in additional second-round reads for RAMP-F and RAMP-H. In contrast, lock-based algorithms fare poorly under high contention, while E-PCI indirected commit checks again incurred additional overhead. By relying on clients (rather than additional partitions) to repair fractured writes, RAMP-F, RAMP-H, and RAMP-S performance is less affected by hot items.

Overall, RAMP-F and RAMP-H exhibit performance close to that of no concurrency control due to their independence properties and guaranteed worst-case performance. As the proportion of writes

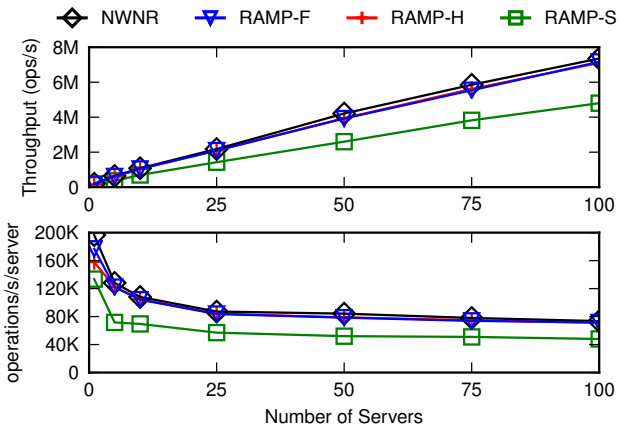


**Figure 4: RAMP transactions scale linearly to over 7 million operations/s with comparable performance to NWNR baseline.**

increases, an increasing proportion of RAMP-F and RAMP-H operations take two RTTs and performance trends towards that of RAMP-S, which provides a constant two RTT overhead. In contrast, lock-based protocols perform poorly under contention while E-PCI triggers more commit checks than RAMP-F and RAMP-H trigger second round reads (but still performs well without contention and for particularly read-heavy workloads). The ability to allow clients to independently verify read sets enables good performance despite a range of (sometimes adverse) conditions (e.g., high contention).

## 5.3 Experimental Results: CTP Overhead

We also evaluated the overhead of blocked writes in our implementation of the Cooperative Termination Protocol discussed in Section 4.6. To simulate blocked writes, we artificially dropped a percentage of COMMIT commands in PUT_ALL calls such that clients returned from writes early and partitions were forced to complete the commit via CTP. This behavior is worse than expected because "blocked" clients continue to issue new operations. The table below reports the throughput reduction as the proportion of blocked writes increases (compared to no blocked writes) for a workload of 100% RAMP-F write transactions:

| **Blocked %** | 0.01% | 0.1% | 25% | 50% |
|---|---|---|---|---|
| **Throughput** | No change | 99.86% | 77.53% | 67.92% |

As these results demonstrate, CTP can reduce throughput because each commit check consumes resources (here, network and CPU capacity). However, CTP only performs commit checks in the event of blocked writes (or time-outs; set to 5s in our experiments), so a modest failure rate of 1 in 1000 writes has a limited effect. The higher failure rates produce a near-linear throughput reduction but, in practice, a blocking rate of even a few percent is likely indicative of larger systemic failures. As Figure 3 hints, the effect of additional metadata for the participant list in RAMP-H and RAMP-S is limited, and, for our default workload of 5% writes, we observe similar trends but with throughput degradation of 10% or less across the above configurations. This validates our initial motivation behind the choice of CTP: average-case overheads are small.

## 5.4 Experimental Results: Scalability

We finally validate our chosen scalability criteria by demonstrating linear scalability of RAMP transactions to 100 servers. We deployed an increasing number of servers within the us-west-2 EC2 region and, to mitigate the effects of hot items during scaling, configured uniform random access to items. We were unable to include more than 20 instances in an EC2 "placement group," which guarantees 10 GbE connections between instances, so, past 20 servers, servers communicated over a degraded network. Around 40 servers, we exhausted the us-west-2b "availability zone" (datacenter) capacity and had to allocate our instances across the remaining zones, further degrading network performance. However, as shown in Figure 4, each RAMP algorithm scales linearly, even though in expectation, at 100 servers, all but one in 100M transactions is a multi-partition operation. In particular, RAMP-F achieves slightly under 7.1 million operations per second, or 1.79 million transactions per second on a set of 100 servers (71,635 operations per partition per second). At all scales, RAMP-F throughput was always within 10% of NWNR. With 100 servers, RAMP-F was within 2.6%, RAMP-S within 3.4%, and RAMP-S was within 45% of NWNR. In light of our scalability criteria, this behavior is unsurprising.

# 6. RELATED WORK

Replicated databases offer a broad spectrum of isolation guarantees at varying costs to performance and availability [8]:

**Serializability.** At the strong end of the isolation spectrum is serializability, which provides transactions with the equivalent of a serial execution (and therefore also provides RA). A range of techniques can enforce serializability in distributed databases [3, 8], multi-version concurrency control (e.g. [37]) locking (e.g. [31]), and optimistic concurrency control [41]. These useful semantics come with costs in the form of decreased concurrency (e.g., contention and/or failed optimistic operations) and limited availability during partial failure [5, 21]. Many designs [19, 29] exploit cheap serializability within a single partition but face scalability challenges for distributed operations. Recent industrial efforts like F1 [41] and Spanner [17] have improved performance via aggressive hardware advances but, their reported throughput is still limited to 20 and 250 writes per item per second. Multi-partition serializable transactions are expensive and, especially under adverse conditions, are likely to remain expensive [18, 28, 35].

**Weak isolation.** The remainder of the isolation spectrum is more varied. Most real-world databases offer (and often default to) non-serializable isolation models [5, 34]. These "weak isolation" levels allow greater concurrency and fewer system-induced aborts compared to serializable execution but provide weaker semantic guarantees. For example, the popular choice of Snapshot Isolation prevents Lost Update anomalies but not Write Skew anomalies [2]; by preventing Lost Update, concurrency control mechanisms providing Snapshot Isolation violate synchronization independence [5]. In recent years, many "NoSQL" designs have avoided cross-partition transactions entirely, effectively providing Read Uncommitted isolation in many industrial databases such PNUTS [15], Dynamo [22], TAO [11], Espresso [38], Rainbird [44], and BigTable [13]. These systems avoid penalties associated with stronger isolation but in turn sacrifice transactional guarantees (and therefore do not offer RA).

**Related mechanisms.** There are several algorithms that are closely related to our choice of RA and RAMP algorithm design.

COPS-GT's two-round read-only transaction protocol [32] is similar to RAMP-F reads—client read transactions identify causally inconsistent versions by timestamp and fetch them from servers. While COPS-GT provides causal consistency (requiring additional metadata), it does not support RA isolation for multi-item writes.

Eiger provides its write-only transactions [33] by electing a coordinator server for each write. As discussed in Section 5 (E-PCI), the number of "commit checks" performed during its read-only transactions is proportional to the number of concurrent writes. Using a coordinator violates partition independence but in turn provides

causal consistency. This coordinator election is analogous to G-Store's dynamic key grouping [19] but with weaker isolation guarantees; each coordinator effectively contains a partitioned completed transaction list from [12]. Instead of relying on indirection, RAMP transaction clients autonomously assemble reads and only require constant factor (or, for RAMP-F, linear in transaction size) metadata size compared to Eiger's *PL-2L* (worst-case linear in database size).

RAMP transactions are inspired by our earlier proposal for *Monotonic Atomic View* (MAV) isolation: transactions read from a monotonically advancing view of database state [5]. MAV is strictly weaker than RA and does not prevent fractured reads, as required for our applications (i.e., reads are not guaranteed to be transactionally aligned). The prior MAV algorithm we briefly sketched in [5] is similar to RAMP-F but, as a consequence of its weaker semantics, allows one-round read transactions. The RAMP algorithms described here are portable to the highly available (i.e., non-linearizable, "AP/EL" [1, 23]) replicated setting of [5], albeit with necessary penalties to latency between updates and their visibility.

Overall, we are not aware of a concurrency control mechanism for partitioned databases that provides synchronization independence, partition independence, and at least RA isolation.

# 7. CONCLUSION

This paper described how to achieve atomically visible multi-partition transactions without incurring the performance and availability penalties of traditional algorithms. We first identified a new isolation level—Read Atomic isolation—that provides atomic visibility and matches the requirements of a large class of real-world applications. We subsequently achieved RA isolation via scalable, contention-agnostic RAMP transactions. In contrast with techniques that use inconsistent but fast updates, RAMP transactions provide correct semantics for applications requiring secondary indexing, foreign key constraints, and materialized view maintenance while maintaining scalability and performance. By leveraging multi-versioning with a variable but small (and, in two of three algorithms, constant) amount of metadata per write, RAMP transactions allow clients to detect and assemble atomic sets of versions in one to two rounds of communication with servers (depending on the RAMP implementation). The choice of synchronization and partition independent algorithms allowed us to achieve near-baseline performance across a variety of workload configurations and scale linearly to 100 servers. While RAMP transactions are not appropriate for all applications, the many for which they are well suited will benefit measurably.

**Acknowledgments** The authors would like to thank Peter Alvaro, Giselle Cheung, Neil Conway, Aaron Davidson, Mike Franklin, Aurojit Panda, Nuno Preguiça, Edward Ribeiro, Shivaram Venkataraman, and the SIGMOD reviewers for their insightful feedback. This research is supported by NSF CISE Expeditions award CCF-1139158 and DARPA XData Award FA8750-12-2-0331, the National Science Foundation Graduate Research Fellowship (grant DGE-1106400), and gifts from Amazon Web Services, Google, SAP, Apple, Inc., Cisco, Clearstory Data, Cloudera, EMC, Ericsson, Facebook, GameOnTalis, General Electric, Hortonworks, Huawei, Intel, Microsoft, NetApp, NTT Multimedia Communications Laboratories, Oracle, Samsung, Splunk, VMware, WANdisco and Yahoo!.

# 8. REFERENCES

[1] D. J. Abadi. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *IEEE Computer*, 45(2):37–42, 2012.

[2] A. Adya. *Weak consistency: a generalized theory and optimistic implementations for distributed transactions*. PhD thesis, MIT, 1999.

[3] D. Agrawal and V. Krishnaswamy. Using multiversion data for non-interfering execution of write-only transactions. In *SIGMOD 1991*.

[4] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics (2nd edition)*. John Wiley Interscience, March 2004.

[5] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Highly Available Transactions: Virtues and Limitations. In *VLDB 2014*.

[6] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. The potential dangers of causal consistency and an explicit solution. In *SOCC 2012*.

[7] J. Baker, C. Bond, J. Corbett, J. Furman, et al. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR 2011*.

[8] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Addison-wesley New York, 1987.

[9] K. Birman, G. Chockler, and R. van Renesse. Toward a cloud computing research agenda. *SIGACT News*, 40(2):68–80, June 2009.

[10] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *CACM*, 13(7):422–426, 1970.

[11] N. Bronson, Z. Amsden, G. Cabrera, P. Chukka, P. Dimov, et al. TAO: Facebook's distributed data store for the social graph. In *USENIX ATC 2013*.

[12] A. Chan and R. Gray. Implementing distributed read-only transactions. *IEEE Transactions on Software Engineering*, (2):205–212, 1985.

[13] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, et al. Bigtable: A distributed storage system for structured data. In *OSDI 2006*.

[14] R. Chirkova and J. Yang. Materialized views. *Foundations and Trends in Databases*, 4(4):295–405, 2012.

[15] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, et al. PNUTS: Yahoo!'s hosted data serving platform. In *VLDB 2008*.

[16] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *ACM SOCC 2010*.

[17] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, et al. Spanner: Google's globally-distributed database. In *OSDI 2012*.

[18] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. In *VLDB 2010*.

[19] S. Das, D. Agrawal, and A. El Abbadi. G-store: a scalable data store for transactional multi key access in the cloud. In *ACM SOCC 2010*.

[20] K. Daudjee and K. Salem. Lazy database replication with ordering guarantees. In *ICDE 2004*, pages 424–435.

[21] S. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in partitioned networks. *ACM Computing Surveys*, 17(3):341–370, 1985.

[22] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, et al. Dynamo: Amazon's highly available key-value store. In *SOSP 2007*.

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

[24] J. Gray and L. Lamport. Consensus on transaction commit. *ACM TODS*, 31(1):133–160, Mar. 2006.

[25] P. Helland. Life beyond distributed transactions: an apostate's opinion. In *CIDR 2007*.

[26] S. Hull. 20 obstacles to scalability. *Commun. ACM*, 56(9):54–59, 2013.

[27] N. Huyn. Maintaining global integrity constraints in distributed databases. *Constraints*, 2(3/4):377–399, Jan. 1998.

[28] E. P. Jones, D. J. Abadi, and S. Madden. Low overhead concurrency control for partitioned main memory databases. In *SIGMOD 2010*.

[29] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, et al. H-Store: a high-performance, distributed main memory transaction processing system. In *VLDB 2008*.

[30] R. J. Lipton and J. S. Sandberg. PRAM: a scalable shared memory. Technical Report TR-180-88, Princeton University, September 1988.

[31] F. Llirbat, E. Simon, D. Tombroff, et al. Using versions in update transactions: Application to integrity checking. In *VLDB 1997*.

[32] W. Lloyd, M. J. Freedman, et al. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *SOSP 2011*.

[33] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *NSDI 2013*.

[34] C. Mohan. History repeats itself: Sensible and NonsenSQL aspects of the NoSQL hoopla. In *EDBT 2013*.

[35] A. Pavlo, C. Curino, and S. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *SIGMOD 2012*.

[36] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *OSDI 2010*.

[37] S. H. Phatak and B. Badrinath. Multiversion reconciliation for mobile databases. In *ICDE 1999*.

[38] L. Qiao, K. Surlaker, S. Das, T. Quiggle, et al. On brewing fresh Espresso: LinkedIn's distributed data serving platform. In *SIGMOD 2013*.

[39] N. Schiper, P. Sutra, and F. Pedone. P-store: Genuine partial replication in wide area networks. In *IEEE SRDS 2010*.

[40] M. Shapiro et al. A comprehensive study of convergent and commutative replicated data types. Technical Report 7506, INRIA, 2011.

[41] J. Shute et al. F1: A distributed SQL database that scales. In *VLDB 2013*.

[42] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch. Session guarantees for weakly consistent replicated data. In *PDIS 1994*.

[43] A. Thomson, T. Diamond, S. Weng, K. Ren, P. Shao, and D. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *SIGMOD 2012*.

[44] K. Weil. Rainbird: Real-time analytics at Twitter. Strata 2011 http://slidesha.re/hjMOui.

[45] S. B. Zdonik. Object-oriented type evolution. In *DBPL*, pages 277–288, 1987.

[46] J. Zhou et al. Lazy maintenance of materialized views. In *VLDB 2007*.

# APPENDIX: Proofs and Isolation Details

**RAMP-F Correctness.** To prove RAMP-F provides RA isolation, we show that the two-round read protocol returns a transactionally atomic set of versions. To do so, we formalize criteria for atomic (read) sets of versions in the form of *companion sets*. We will call the set of versions produced by a transaction *sibling versions* and call two items from the same write set *sibling items*.

Given two versions $x_i$ and $y_j$, we say that $x_i$ is a *companion* to $y_j$ if $x_i$ is a transactional sibling of $y_j$ or $x$ is a sibling item of $y_j$ and $i > j$. We say that a set of versions $V$ is a *companion set* if, for every pair $(x_i, y_j)$ of versions in $V$ where $x$ is a sibling item of $y_j$, $x_i$ is a companion to $y_j$. In Figure 1, the versions returned by $T_2$'s first round of reads ($\{x_1, y_\perp\}$) do not comprise a companion set because $y_\perp$ has a lower timestamp than $x_1$'s sibling version of $y$ (that is, $x_1$ has sibling version $y_1$ and but $\perp < 1$ so $y_\perp$ has too low of a timestamp). Subsets of companion sets are also companion sets and companion sets also have a useful property for RA isolation:

**Claim 1** (Companion sets are atomic)**.** Companion sets do not contain fractured reads.

*Proof.* Claim 1 follows from the definitions of companion sets and fractured reads. If $V$ is a companion set, then every version $x_i \in V$ is also a companion to every other version $y_j \in V$ where $v_j$ contains $x$ in its sibling items. If $V$ contained fractured reads, $V$ would contain two versions $x_i, y_j$ such that the transaction that wrote $y_j$ also wrote a version $x_k$, $i < k$. However, in this case, $x_i$ would not be a companion to $y_j$, a contradiction. Therefore, $V$ cannot contain fractured reads. □

To provide RA, RAMP-F clients assemble a companion set for the requested items (in $v_{latest}$), which we prove below:

**Claim 2.** RAMP-F provides Read Atomic isolation.

*Proof.* Each write in RAMP-F contains information regarding its siblings, which can be identified by item and timestamp. Given a set of RAMP-F versions, recording the highest timestamped version of each item (as recorded either in the version itself or via sibling metadata) yields a companion set of item-timestamp pairs: if a client reads two versions $x_i$ and $y_j$ such that $x$ is in $y_j$'s sibling items but $i < j$, then $v_{latest}[x]$ will contain $j$ and not $i$. Accordingly, given the versions returned by the first round of RAMP-F reads, clients calculate a companion set containing versions of the requested items. Given this companion set, clients check the first-round versions against this set by timestamp and issue a second round of reads to fetch any companions that were not returned in the first round. The resulting set of versions will be a subset of the computed companion set and will therefore also be a companion set. This ensures that the returned results do not contain fractured reads. RAMP-F first-round reads access *lastCommit*, so each transaction corresponding to a first-round version is committed, and, therefore, any siblings requested in the (optional) second round of reads are also committed. Accordingly, RAMP-F never reads aborted or non-final (intermediate) writes. This establishes that RAMP-F provides RA. □

**RAMP-F Scalability and Independence.** RAMP-F also provides the independence guarantees from Section 3.3. The following invariant over *lastCommit* is core to RAMP-F GET request completion:

**Invariant 1** (Companions present)**.** If a version $x_i$ is referenced by *lastCommit* (that is, $lastCommit[x] = i$), then each of $x_i$'s sibling versions are present in *versions* on their respective partitions.

Invariant 1 is maintained by RAMP-F's two-phase write protocol. *lastCommit* is only updated once a transaction's writes have been placed into *versions* by a first round of PREPARE messages. Siblings will be present in *versions* (but not necessarily *lastCommit*).

**Claim 3.** RAMP-F provides synchronization independence.

*Proof.* Clients in RAMP-F do not communicate or coordinate with one another and only contact servers. Accordingly, to show that RAMP-F provides synchronization independence, it suffices to show that server-side operations always terminate. PREPARE and COMMIT methods only access data stored on the local partition and do not block due to external coordination or other method invocations; therefore, they complete. GET requests issued in the first round of reads have $ts_{req} = \perp$ and therefore will return the version corresponding to $lastCommit[k]$, which was placed into *versions* in a previously completed PREPARE round. GET requests issued in the second round of client reads have $ts_{req}$ set to the client's calculated $v_{latest}[k]$. $v_{latest}[k]$ is a sibling of a version returned from *lastCommit* in the first round, so, due to Invariant 1, the requested version will be present in *versions*. Therefore, GET invocations are guaranteed access to their requested version and can return without waiting. The success of RAMP-F operations do not depend on the success or failure of other clients' RAMP-F operations. □

**Claim 4.** RAMP-F provides partition independence.

*Proof.* RAMP-F transactions do not access partitions that are unrelated to each transaction's specified data items and servers do not contact other servers in order to provide a safe response for operations. □

**RAMP-S Correctness.** RAMP-S writes and first-round reads proceed identically to RAMP-F writes, but the metadata written and returned is different. Therefore, the proof is similar to RAMP-F, with a slight modification for the second round of reads.

**Claim 5.** RAMP-S provides Read Atomic isolation.

*Proof.* To show that RAMP-S provides RA, it suffices to show that RAMP-S second-round reads (*resp*) are a companion set. Given two versions $x_i, y_j \in resp$ such that $x \neq y$, if $x$ is a sibling item of $y_j$, then $x_i$ must be a companion to $y_j$. If $x_i$ were not a companion to $y_j$, then it would imply that $x$ is not a sibling item of $y_j$ (so we are done) or that $j > i$. If $j > i$, then, due to Invariant 1 (which also holds for RAMP-S writes due to identical write protocols), $y_j$'s sibling is present in *versions* on the partition for $x$ and would have been returned by the server (line 6), a contradiction. Each second-round GET request returns only one version, so we are done. □

**RAMP-S Scalability and Independence.** RAMP-S provides synchronization independence and partition independence. For brevity, we again omit full proofs, which closely resemble those of RAMP-F.

**RAMP-H Correctness.** The probabilistic behavior of the RAMP-H Bloom filter admits false positives. However, given unique transaction timestamps (Section 4.5), requesting false siblings by timestamp and item does not affect correctness:

**Claim 6.** RAMP-H provides Read Atomic isolation.

*Proof.* To show that RAMP-H provides Read Atomic isolation, it suffices to show that any versions requested by RAMP-H second-round reads that would not have been requested by RAMP-F second-round reads (call this set $v_{false}$) do not compromise the validity of RAMP-H's returned companion set. Any versions in $v_{false}$ do not exist: timestamps are unique, so, for each version $x_i$, there are no versions $x_j$ of non-sibling items with the same timestamp as $x_i$ (i.e., where $i = j$). Therefore, requesting versions in $v_{false}$ do not change the set of results collected in the second round. □

**RAMP-H Scalability and Independence.** RAMP-H provides synchronization independence and partition independence. We omit full proofs, which closely resemble those of RAMP-F. The only significant difference from RAMP-F is that second-round GET requests may return $\perp$, but, as we showed above, these empty responses correspond to false positives in the Bloom filter and therefore do not affect correctness.

**Comparison to other isolation levels.** The fractured reads anomaly is similar to Adya's "Missing Transaction Updates" definition, only applied to immediate read dependencies (rather than all transitive dependencies). RA is stronger than *PL-2* (Read Committed), but weaker than *PL-SI*, *PL-CS*, and *PL-2.99* (notably, RA does not prevent anti-dependency cycles, or Adya's *G2* or *G-SIa*—informally, it allows concurrent updates) [2].

RA does not (by itself) provide ordering guarantees across transactions. Our RAMP implementations provide a variant of PRAM consistency, where, for each item, each user's writes are serialized [30] (i.e., "session" ordering [20]), and, once a user's operation completes, all other users will observe its effects (regular register semantics, applied at the transaction level). This provides transitivity with respect to each user's operations. For example, if a user updates her privacy settings and subsequently posts a new photo, the photo cannot be read without the privacy setting change [15]. However, PRAM does not respect the *happens-before* relation [4] across users. If Sam reads Mary's comment and replies to it, other users may read Sam's comment without Mary's comment. In this case, RAMP transactions can leverage explicit causality [6] via foreign key dependencies, but happens-before is not provided by default. If required, we believe it is possible to enforce happens-before but, due to scalability concerns regarding metadata and partition independence (e.g., [6] and Section 5), do not further explore this possibility. An "active-active" replicated implementation can provide available [5, 23] operation at the cost of these recency guarantees.