Scalable Atomic Visibility with RAMP Transactions

PETER BAILIS, Stanford University
ALAN FEKETE, University of Sydney
ALI GHODSI, JOSEPH M. HELLERSTEIN, and ION STOICA, UC Berkeley

Databases can provide scalability by partitioning data across several servers. However, multipartition, multioperation 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 multipartition 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 Multipartition (RAMP) transactions that enforce atomic visibility while offering excellent scalability, guaranteed commit despite partial failures (via *coordination-free execution*), 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 multiversioning and by allowing clients to independently resolve nonatomic reads. We demonstrate that, in contrast with existing algorithms, RAMP transactions incur limited overhead—even under high contention—and scale linearly to 100 servers.

Categories and Subject Descriptors: H.2.4 [**Systems**]: Distributed Databases

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Atomic visibility, transaction processing, NoSQL, secondary indexing, materialized views

ACM Reference Format:
Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2016. Scalable atomic visibility with RAMP transactions. ACM Trans. Database Syst. 41, 3, Article 15 (July 2016), 45 pages.
DOI: http://dx.doi.org/10.1145/2909870

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 [Baker et al. 2011; Chang et al. 2006; Curino et al. 2010; Das et al. 2010; DeCandia et al. 2007; Kallman et al. 2008;

This research was 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!.
Authors' addresses: P. Bailis, 353 Serra Mall, Stanford University, Stanford, CA 94305; email: pbailis@cs.stanford.edu; A. Fekete, School of Information Technologies, building J12, University of Sydney, NSW 2006, Australia; email: alan.fekete@sydney.edu.au; A. Ghodsi, J. M. Hellerstein, and I. Stoica, 387 Soda Hall, Berkeley, CA 94720; emails: {alig, hellerstein, istoica}@cs.berkeley.edu.

DOI: http://dx.doi.org/10.1145/2909870

Thomson et al. 2012]. 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 [Corbett et al. 2012; Kallman et al. 2008; Jones et al. 2010; Pavlo et al. 2012].

In this work, we address a largely underserved class of applications requiring multipartition, atomically visible[1] *transactional* access: cases where all or none of each transaction's effects should be visible. The status quo for these multipartition 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 [Bronson et al. 2013; Chang et al. 2006; Cooper et al. 2008; DeCandia et al. 2007; Hull 2013; Qiao et al. 2013; Weil 2011]. This may lead to anomalous behavior for several use cases requiring 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 [Bernstein et al. 1987; Corbett et al. 2012; Thomson et al. 2012]. 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 [Bailis et al. 2014a; Curino et al. 2010; Jones et al. 2010; Pavlo et al. 2012]. This dichotomy between scalability and atomic visibility has been described as "a fact of life in the big cruel world of huge systems" [Helland 2007]. The proliferation of nontransactional multi-item operations is symptomatic of a widespread "fear of synchronization" at scale [Birman et al. 2009].

Our contribution in this article 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, nonserializable 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 Multipartition (RAMP) transactions (Section 4). RAMP transactions guarantee scalability and outperform existing atomic algorithms because they satisfy two key scalability constraints. First, RAMP transactions guarantee *coordination-free execution*: one client's transactions cannot cause another client's transactions to stall or fail. Second, RAMP transactions guarantee *partition independence*: clients only contact partitions that their transactions directly reference (i.e., there is no central master, coordinator, or scheduler). Together, these properties ensure limited coordination across partitions and horizontal scalability for multipartition 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 nonatomic (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 multiversioning 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

[1]Our use of "atomic" (specifically, RA 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 [Bernstein et al. 1987] or linearizability [Attiya and Welch 2004].

(a timestamp per write) and two RoundTrip 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 [Bloom 1970] to provide an intermediate solution. Traditional techniques like locking 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 [Bailis et al. 2014a; Adya 1999], 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.

The remainder of this article proceeds as follows: Section 2 presents an overview of RAMP transactions and describes key use cases based on industry reports. Section 3 defines RA isolation, presents both a detailed comparison with existing isolation guarantees and a syntactic condition, the Read-Subset-Writes property, that guarantees equivalence to serializable isolation, and defines two key scalability criteria for RAMP algorithms to provide. Section 4 presents and analyzes three RAMP algorithms, which we experimentally evaluate in Section 5. Section 6 presents modifications of the RAMP protocols to better support multidatacenter deployments and to enforce transitive dependencies. Section 7 compares with Related Work, and Section 8 concludes with a discussion of promising future extensions to the protocols presented here.

2. OVERVIEW AND MOTIVATION

In this article, 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 x and y are initially *null* and 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 RA 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 article 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 [Bernstein et al. 1987]. (Optimistic concurrency control mechanisms can similarly block during validation.) We instead use ACPs with nonblocking 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' nonblocking behavior means that they cannot provide certain guarantees like preventing concurrent updates. However, applications that can use RA 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. RA Isolation in the Wild

As a simple example, consider a social networking application: if two users, Sam and Mary, become "friends" (a bidirectional 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 RA isolation would correctly enforce this behavior. We can further classify three general use cases for RA 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 [Bronson et al. 2013], LinkedIn's Espresso [Qiao et al. 2013], and Yahoo! PNUTS [Cooper et al. 2008] store information about business entities such as users, photos, and status updates as well as relationships between them (e.g., the previous friend relationships). Their data models often represent bidirectional edges as two distinct unidirectional 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 [Bronson et al. 2013]. PNUTS's authors describe an identical scenario [Cooper et al. 2008]. These applications require foreign key maintenance and often, due to their unidirectional relationships, multientity update and access. Violations of atomic visibility surface as broken bidirectional relationships (as with Sam and Mary previously) and dangling or incorrect references. For example, clients should never observe that 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 avoid dangling pointers by creating a "tombstone" at the opposite end of the association (i.e., delete any entries with associations via a special record that signifies deletion) [Zdonik 1987].

(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 challenging (e.g., indexing by birth date). There are two dominant strategies for distributed secondary indexing. First, the *local secondary index* approach colocates secondary indexes and

primary data, so each server contains a secondary index that only references and indexes data stored on its server [Baker et al. 2011; Qiao et al. 2013]. 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 [Bronson et al. 2013; Corbett et al. 2012; Qiao et al. 2013]. Alternatively, the *global secondary index* approach locates secondary indexes (which may be partitioned, but by a secondary attribute) separately from primary data [Cooper et al. 2008; Baker et al. 2011]. This alternative allows fast secondary lookups (read-one) but requires multipartition update (at least write-two).

Real-world services employ either local secondary indexing (e.g., Espresso [Qiao et al. 2013], Cassandra, and Google Megastore's local indexes [Baker et al. 2011]) or nonatomic (incorrect) global secondary indexing (e.g., Espresso and Megastore's global indexes, Yahoo! PNUTS's proposed secondary indexes [Cooper et al. 2008]). The former is nonscalable 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, suppose 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`). 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 [Weil 2011], Google's Percolator [Peng and Dabek 2010], and LinkedIn's Espresso systems [Qiao et al. 2013]. 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., prematerialize) a count of unread messages rather than scan all messages every time a user accesses her mailbox [Qiao et al. 2013]. 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 60 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 [Chirkova and Yang 2012; Huyn 1998; Blakeley et al. 1986], 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 [Zhou et al. 2007]). For more complex views, such as counters, users can execute RAMP transactions over specialized data structures such as the CRDT G-Counter [Shapiro et al. 2011].

In brief: Status Quo. Despite application requirements for RA 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 [Bronson et al. 2013; Qiao et al. 2013; Cooper et al. 2008]. These systems are not exceptions: data stores like Bigtable [Chang et al. 2006], Dynamo [DeCandia et al. 2007], and many popular "NoSQL" [Mohan 2013] and even some "NewSQL" [Bailis et al. 2014a] stores do not provide transactional guarantees for multi-item operations. Unless users are willing to sacrifice scalability by opting for serializable semantics [Corbett et al. 2012], they are often left without transactional semantics.

The designers of these Internet-scale, real-world systems have made a conscious decision to provide scalability at the expense of multipartition 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 RA isolation and, to capture scalability, formulate a pair of strict scalability criteria: coordination-free execution 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 [Adya 1999; Bernstein et al. 1987], 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 *item read set* and *item write set*. Each write creates a *version* of an item and we identify versions of items by a *timestamp* taken from a totally ordered set (e.g., natural numbers) that is unique across all versions of each item. Timestamps therefore induce a total order on versions of each item, and we denote version i of item x as x_i. All items have an initial version $\perp$ that is located at the start of each order of versions for each item and is produced by an initial transaction $T_\perp$. Each transaction ends in a *commit* or an *abort* operation; we call a transaction that commits a *committed* transaction and a transaction that aborts an *aborted* transaction. In our model, we consider *histories* comprised of a set of transactions along with their read and write operations, versions read and written, and commit or abort operations. In our example histories, all transactions commit unless otherwise noted.

Definition 3.1 (*Fractured Reads*). A transaction T_j exhibits the *fractured reads* phenomenon if transaction T_i writes versions x_a and y_b (in any order, where x and y may or may not be distinct items), T_j reads version x_a and version y_c, and $c < b$.

We also define RA isolation to prevent transactions from reading uncommitted or aborted writes. This is needed to capture the notion that, under RA isolation, readers only observe the final output of a given transaction that has been accepted by the database. To do so, we draw on existing definitions from the literature on weak isolation. Though these guarantees predate Adya's dissertation [Adya 1999], we use his formalization of them, which, for completeness, we reproduce in the following in the context of our system model. We provide some context for the interested reader, however Adya [1999] provides the most comprehensive treatment.

Read dependencies capture behavior where one transaction observes another transaction's writes.

Definition 3.2 (*Read-Depends*). Transaction T_j *directly read-depends* on T_i if transaction T_i writes some version x_i and T_j reads x_i.

Antidependencies capture behavior where one transaction overwrites the versions that another transaction reads. In a multiversioned model like Adya's, we define overwrites according to the version order defined for that item.

Definition 3.3 (*Antidepends*). Transaction T_j *directly antidepends* on T_i if transaction T_i reads some version x_k and T_j writes x's next version (after x_k) in the version order. Note that the transaction that wrote the later version directly antidepends on the transaction that read the earlier version.

Write dependencies capture behavior where one transaction overwrites another transaction's writes.

Definition 3.4 (*Write-Depends*). Transaction T_j *directly write-depends* on T_i if T_i writes a version x_i and T_j writes x's next version (after x_i) in the version order.

We can combine these three kinds of labeled edges into a data structure called the Direct Serialization Graph.

Definition 3.5 (*Direct Serialization Graph*). We define the Direct Serialization Graph (DSG) arising from a history H, denoted by $DSG(H)$ as follows. Each node in the graph corresponds to a committed transaction and directed edges correspond to different types of direct conflicts. There is a *read dependency edge, write dependency edge, or antidependency edge* from transaction T_i to transaction T_j if T_j *reads*/*writes*/*directly antidepends* on T_i.

We can subsequently define several kinds of undesirable behaviors by stating properties about the DSG. The first captures the intent of the Read Uncommitted isolation level [Berenson et al. 1995].

Definition 3.6 (*G0: Write Cycles*). A history H exhibits phenomenon *G0* if $DSG(H)$ contains a directed cycle consisting entirely of write-dependency edges.

The next three undesirable behaviors comprise the intent of the Read Committed isolation level [Berenson et al. 1995].

Definition 3.7 (*G1a: Aborted Reads*). A history H exhibits phenomenon *G1a* if H contains an aborted transaction T_a and a committed transaction T_c such that T_c reads a version written by T_a.

Definition 3.8 (*G1b: Intermediate Reads*). A history H exhibits phenomenon *G1b* if H contains a committed transaction T_i that reads a version of an object x_j written by transaction T_f, and T_f also wrote a version x_k such that $j < k$.

The definition of the Fractured Reads phenomenon subsumes the definition of G1b. For completeness, and, to prevent confusion, we include it here and in our discussion in the following.

Definition 3.9 (*G1c: Circular Information Flow*). A history H exhibits phenomenon *G1c* if $DSG(H)$ contains a directed cycle that consists entirely of read-dependency and write-dependency edges.

Our RAMP protocols prevent G1c by assigning the final write to each item in each transaction the same timestamp. However, to avoid further confusion between the standard practice of assigning each final write in a serializable multiversion history the same timestamp [Bernstein et al. 1987] and the flexibility of timestamp assignment admitted in Adya's formulation of weak isolation, we continue with the previous definitions.

As Adya describes, the previous criteria prevent readers from observing *uncommitted* versions (i.e., those produced by a transaction that has not committed or aborted), *aborted* versions (i.e., those produced by a transaction that has aborted), or *intermediate* versions (i.e., those produced by a transaction but were later overwritten by writes to the same items by the same transaction).

We can finally define RA isolation:

Definition 3.10 (*RA*). A system provides *RA* isolation if it prevents fractured reads phenomena and also proscribes phenomena *G0*, *G1a*, *G1b*, *G1c* (i.e., prevents transactions from reading uncommitted, aborted, or intermediate versions).

Thus, RA informally provides transactions with a "snapshot" view of the database that respects transaction boundaries (see Sections 3.3 and 3.4 for more details, including a discussion of transitivity). RA is simply a restriction on write *visibility*—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; that is, under RA, two transactions are never prevented from both producing different versions of the same 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 preventing 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.

In Section 3.3, we describe RA's relation to other formally defined isolation levels, and in Section 3.4, we discuss when RA provides serializable outcomes.

3.3. RA Compared to Other Isolation Models

In this section, we illustrate RA's relationship to alternative weak isolation models by both example and reference to particular isolation phenomena drawn from Adya [1999] and Bailis et al. [2014a]. We have included it in the extended version of this work in response to valuable reader and industrial commentary requesting clarification on exactly which phenomena RA isolation does and does not prevent.

For completeness, we first reproduce definitions of existing isolation models (again, using Adya's models) and then give example histories to compare and contrast with RA isolation.

Read Committed. We begin with the common [Bailis et al. 2014a] Read Committed isolation. Note that the phenomena mentioned in the following were defined in Section 3.1.

Definition 3.11 (*Read Committed* (*PL-2 or RC*)). Read Committed isolation proscribes phenomena *G1a*, *G1b*, *G1c*, and *G0*.

RA is stronger than Read Committed as Read Committed does not prevent fractured reads. History 1 does not respect RA isolation. After T_1 commits, both T_2 and T_3 could both commit but, to prevent fractured reads, T_4 and T_5 must abort. History (1) respects

RC isolation and all transactions can safely commit.

$$\begin{aligned} T_1 &\ w(x_1); w(y_1), \\ T_2 &\ r(x_\perp); r(y_\perp), \\ T_3 &\ r(x_1); r(y_1), \\ T_4 &\ r(x_\perp); r(y_1), \\ T_5 &\ r(x_1); r(y_\perp). \end{aligned} \tag{1}$$

Lost Updates. Lost Updates phenomena informally occur when two transactions simultaneously attempt to make conditional modifications to the same data item(s).

Definition 3.12 (*Lost Update*). A history H exhibits the phenomenon Lost Updates if $DSG(H)$ contains a directed cycle having one or more antidependency edges and all edges are by the same data item x.

That is, Lost Updates occur when the version that a transaction reads is overwritten by a second transaction that the first transaction depends on. History (2) exhibits the Lost Updates phenomenon: T_1 antidepends on T_2 because T_2 overwrites $x_\perp$, which T_1 read, and T_2 antidepends on T_1 because T_1 also overwrites $x_\perp$, which T_2 read. However, RA does not prevent Lost Updates phenomena, and History (2) is valid under RA. That is, T_1 and T_2 can both commit under RA isolation.

$$\begin{aligned} T_1 &\ r(x_\perp); w(x_1), \\ T_2 &\ r(x_\perp); w(x_2). \end{aligned} \tag{2}$$

History (2) is invalid under a stronger isolation model that prevents Lost Updates phenomena. For completeness, we provide definitions of two such models—Snapshot Isolation and Cursor Isolation—in the following.

Informally, a history is snapshot isolated if each transaction reads a transaction-consistent snapshot of the database and each of its writes to a given data item x is the first write in the history after the snapshot that updates x. To more formally describe Snapshot Isolation, we formalize the notion of a predicate-based read, as in Adya. Per Adya, queries and updates may be performed on a set of items if a certain condition called the *predicate* applies. We call the set of items a predicate P refers to as P's *logical range* and denote the set of versions returned by a predicate-based read $r_j(P_j)$ as $Vset(P_j)$. We say a transaction T_i changes the matches of a predicate-based read $r_j(P_j)$ if T_i overwrites a version in $Vset(P_j)$.

Definition 3.13 (*Version Set of a Predicate-Based Operation*). When a transaction executes a read or write based on a predicate P, the system selects a version for each item to which P applies. The set of selected versions is called the *Version set* of this predicate-based operation and is denoted by $Vset(P)$.

Definition 3.14 (*Predicate-Many-Preceders (PMP)*). A history H exhibits phenomenon PMP if, for all predicate-based reads $r_i(P_i : Vset(P_i))$ and $r_j(P_j : Vset(P_j)$ in T_k such that the logical ranges of P_i and P_j overlap (call it P_o) and the set of transactions that change the matches of $r_i(P_i)$ and $r_j(P_j)$ for items in P_o differ.

To complete our definition of Snapshot Isolation, we must also consider a variation of the *DSG*. Adya describes the Unfolded Serialization Graph (USG) in Adya [1999, Section 4.2.1]. For completeness, we reproduce it here. The USG is specified for a transaction of interest, T_i, and a history, H, and is denoted by $USG(H, T_i)$. For the USG, we retain all nodes and edges of the *DSG* except for T_i and the edges incident on it. Instead, we split the node for T_i into multiple nodes—one node for every read/write event in T_i. The edges are now incident on the relevant event of T_i.

$USG(H, T_i)$ is obtained by transforming $DSG(H)$ as follows: For each node p ($p \neq T_i$) in $DSG(H)$, we add a node to $USG(H, T_i)$. For each edge from node p to node q in $DSG(H)$, where p and q are different from T_i, we draw a corresponding edge in $USG(H, T_i)$. Now we add a node corresponding to every read and write performed by T_i. Any edge that was incident on T_i in the *DSG* is now incident on the relevant event of T_i in the *USG*. Finally, consecutive events in T_i are connected by *order edges*, e.g., if an action (e.g., SQL statement) reads object y_j and immediately follows a write on object x in transaction T_i, we add an order edge from $w_i(x_i)$ to $r_i(y_j)$.

Definition 3.15 (*Observed Transaction Vanishes (OTV)*). A history H exhibits phenomenon OTV if $USG(H)$ contains a directed cycle consisting of exactly one read-dependency edge by x from T_j to T_i and a set of edges by y containing at least one antidependency edge from T_i to T_j and T_j's read from y precedes its read from x.

Informally, OTV occurs when a transaction observes part of another transaction's updates but not all of them (e.g., T_1 writes x_1 and y_1 and T_2 reads x_1 and $y_\perp$). OTV is weaker than fractured reads because it allows transactions to read multiple versions of the same item: in our previous example, if another transaction T_3 also wrote x_3, T_1 could read x_1 and subsequently x_3 without exhibiting OTV (but this would exhibit fractured reads and therefore violate RA).

With these definitions in hand, we can finally define Snapshot Isolation.

Definition 3.16 (*Snapshot Isolation*). A system provides Snapshot Isolation if it prevents phenomena G0, G1a, G1b, G1c, PMP, OTV, and Lost Updates.

Informally, cursor stability ensures that, while a transaction is updating a particular item x in a history H, no other transactions will update x. We can also formally define Cursor Stability per Adya.

Definition 3.17 (*G-cursor(x)*). A history H exhibits phenomenon *G-cursor(x)* if $DSG(H)$ contains a cycle with an antidependency and one or more write-dependency edges such that all edges correspond to operations on item x.

Definition 3.18 (*Cursor Stability*). A system provides Cursor Stability if it prevents phenomena G0, G1a, G1b, G1c, and G-cursor(x) for all x.

Under either Snapshot Isolation or Cursor Stability, in History (2), either T_1 or T_2, or both would abort. However, Cursor Stability does not prevent fractured reads phenomena, so RA and Cursor Stability are incomparable.

Write Skew. Write Skew phenomena informally occur when two transactions simultaneously attempted to make disjoint conditional modifications to the same data items.

Definition 3.19 (*Write Skew (Adya G2-item)*). A history H exhibits phenomenon Write Skew if $DSG(H)$ contains a directed cycle having one or more antidependency edges.

RA does not prevent Write Skew phenomena. History 3 exhibits the Write Skew phenomenon (Adya's *G2*): T_1 antidepends on T_2 and T_2 antidepends on T_1. However, History 3 is valid under RA. That is, T_1 and T_2 can both commit under RA isolation.

$$\begin{aligned} T_1 \;\; & r(y_\perp); w(x_1), \\ T_2 \;\; & r(x_\perp); w(y_2). \end{aligned} \tag{3}$$

History (3) is invalid under a stronger isolation model that prevents Write Skew phenomena. One stronger model is Repeatable Read, defined next.

Definition 3.20 (*Repeatable Read*). A system provides Repeatable Read isolation if it prevents phenomena *G*0, *G*1*a*, *G*1*b*, *G*1*c*, and Write Skew for nonpredicate reads and writes.

Under Repeatable Read isolation, the system would abort either T_1, T_2, or both. Adya's formulation of Repeatable Read is considerably stronger than the ANSI SQL standard specification [Bailis et al. 2014a].

Missing Dependencies. Notably, RA does not—on its own—prevent missing dependencies or missing transitive updates. We reproduce Adya's definitions as follows:

Definition 3.21 (*Missing Transaction Updates*). A transaction T_j misses an effect of a transaction T_i if T_i writes x_i and commits and T_j reads another version x_k such that $k < i$; that is, T_j reads a version of x that is older than the version that was committed by T_i.

Adya subsequently defines a criterion that prohibits missing transaction updates across all types of dependency edges:

Definition 3.22 (*No-Depend-Misses*). If transaction T_j depends on transaction T_i, T_j does not miss the effects of T_i.

History (4) fails to satisfy No-Depend-Misses but is still valid under RA. That is, T_1, T_2, and T_3 can all commit under RA isolation. Thus, fractured reads prevention is similar to No-Depend-Misses but only applies to immediate read dependencies (rather than all transitive dependencies).

$$\begin{aligned} T_1 &\ \ w(x_1); w(y_1), \\ T_2 &\ \ r(y_1); w(z_2), \\ T_3 &\ \ r(x_\perp); r(z_2). \end{aligned} \tag{4}$$

History (4) is invalid under a stronger isolation model that prevents missing dependencies phenomena, such as standard semantics for Snapshot Isolation (notably, not Parallel Snapshot Isolation [Sovran et al. 2011]) and Repeatable Read isolation. Under these models, the system would abort either T_3 or all of T_1, T_2, and T_3.

This behavior is particularly important and belies the promoted use cases that we discuss in Sections 3.2 and 3.4: writes that should be read together should be written together.

We further discuss the benefits and enforcements of transitivity in Section 6.3.

Predicates. Thus far, we have not extensively discussed the use of predicate-based reads. As Adya notes [Adya 1999] and we described earlier, predicate-based isolation guarantees can be cast as an extension of item-based isolation guarantees (see also Adya's *PL-2L*, which closely resembles RA). RA isolation is no exception to this rule.

Relating to Additional Guarantees. RA isolation subsumes several other useful guarantees. RA prohibits Item-Many-Preceders and Observed Transaction Vanishes phenomena; RA also guarantees Item Cut Isolation, and with predicate support, RA subsumes Predicate Cut Isolation [Bailis et al. 2014b]. Thus, it is a combination of Monotonic Atomic View and Item Cut Isolation. For completeness, we reproduce these definitions next.

Definition 3.23 (*Item-Many-Preceders (IMP)*). A history H exhibits phenomenon IMP if $DSG(H)$ contains a transaction T_i such that T_i directly item-read-depends by x on more than one other transaction.

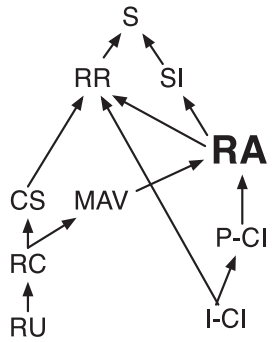


Fig. 1. Comparison of RA with isolation levels from Adya [1999] and Bailis et al. [2014a]. RU: Read Uncommitted, RC: Read Committed, CS: Cursor Stability, MAV: Monotonic Atomic View, ICI: Item Cut Isolation, PCI: Predicate Cut Isolation, RA: Read Atomic, SI: Snapshot Isolation, RR: Repeatable Read (Adya *PL-2.99*), S: Serializable.

Informally, IMP occurs if a transaction observes multiple versions of the same item (e.g., transaction T_i reads x_1 and x_2).

Definition 3.24 (*Item Cut Isolation (I-CI)*). A system that provides Item Cut Isolation prohibits the phenomenon IMP.

Definition 3.25 (*Predicate-Many-Preceders (PMP)*). A history H exhibits the phenomenon PMP if, for all predicate-based reads $r_i(P_i : Vset(P_i))$ and $r_j(P_j : Vset(P_j)$ in T_k such that the logical ranges of P_i and P_j overlap (call it P_o), the set of transactions that change the matches of P_o for r_i and r_j differ.

Informally, PMP occurs if a transaction observes different versions resulting from the same predicate read (e.g., transaction T_i reads $Vset(P_i) = \emptyset$ and $Vset(P_i) = \{x_1\}$)).

Definition 3.26 (*Monotonic Atomic View (MAV)*). A system that provides Monotonic Atomic View isolation prohibits the phenomenon OTV in addition to providing Read Committed isolation.

Summary. Figure 1 relates RA isolation to several existing models. RA is stronger than Read Committed, Monotonic Atomic View, and Cut Isolation; weaker than Snapshot Isolation, Repeatable Read, and Serializability; and incomparable to Cursor Stability.

3.4. RA and Serializability

When we began this work, we started by examining the use cases outlined in Section 2 and derived a weak isolation guarantee that would be sufficient to ensure their correct execution. For general-purpose read-write transactions, RA isolation may indeed lead to nonserializable (and possibly incorrect) database states and transaction outcomes. Yet, as Section 3.2 hints, there appears to be a broader "natural" pattern for which RA isolation appears to provide an intuitive (even "correct") semantics. In this section, we show that for transactions with a particular property of their item read and item write sets, RA is, in fact, serializable. We define this property, called the *Read-Subset-Items-Written (RSIW) property*, prove that transactions obeying the RSIW property lead to serializable outcomes, and discuss the implications of the RSIW property for the applications outlined in Section 2.

Because our system model operates on multiple versions, we must make a small refinement to our use of the term "serializability"—namely, we draw a distinction between serial and one-copy serializable schedules [Bernstein et al. 1987]. First, we say that two histories H_1 and H_2 are *view equivalent* if they contain the same set of

committed transactions and have the same operations and $DSG(H_1)$ and $DSG(H_2)$ have the same direct read dependencies. For brevity, and for consistency with prior work, we say that T_i *reads from* T_j if T_i directly read-depends on T_j. We say that a transaction is *read-only* if it does not contain write operations and that a transaction is *write-only* if it does not contain read operations. In this section, we concern ourselves with *one-copy serializability* [Bernstein et al. 1987], which we define using the previous definition of view equivalence.

Definition 3.27 (*One-Copy Serializability*). A history is one-copy serializable if it is view equivalent to a serial execution of the transactions over a single logical copy of the database.

The basic intuition behind the RSIW property is straightforward: under RA isolation, if application developers use a transaction to bundle a set of writes that should be observed together, any readers of the items that were written will, in fact, behave "properly"—or one-copy serializably. That is, for read-only and write-only transactions, if each reading transaction only reads a subset of the items that another write-only transaction wrote, then RA isolation is equivalent to one-copy serializable isolation. Before proving that this behavior is one-copy serializable, we characterize this condition more precisely:

Definition 3.28 (*RSIW*). A read-only transaction T_r exhibits the *RSIW* property if, whenever T_r reads a version produced by a write-only transaction T_w, T_r only reads items written by T_w.

For example, consider the following History (5):

$$\begin{aligned} T_1 \quad & w(x_1); w(y_1), \\ T_2 \quad & r(x_1); r(y_1), \\ T_3 \quad & r(x_1); r(z_\perp). \end{aligned} \tag{5}$$

Under History (5), T_2 exhibits the RSIW property because it reads a version produced by transaction T_1 and its item read set ($\{x, y\}$) is a subset of T_1's item write set ($\{x, y\}$). However, T_3 does not exhibit the RSIW property because (i) T_3 reads from T_1 but T_3's read set ($\{x, z\}$) is not a subset of T_1's write set ($\{x, y\}$) and (ii) perhaps more subtly, T_3 reads from both T_1 and $T_\perp$.

We say that a history H containing read-only and write-only transactions exhibits the RSIW property (or *has RSIW*) if every read-only transaction in H exhibits the RSIW property.

This brings us to our main result in this section:

THEOREM 3.29. *If a history H containing read-only and write-only transactions has RSIW and is valid under RA isolation, then H is one-copy serializable.*

The proof of Theorem 3.29 is by construction: given a history H has RSIW and is valid under RA isolation, we describe how to derive an equivalent one-copy serial execution of the transactions in H. We begin with the construction procedure, provide examples of how to apply the procedure, then prove that the procedure converts RSIW histories to their one-copy serial equivalents. We include the actual proof in Appendix A.

Utility. Theorem 3.29 is helpful because it provides a simple syntactic condition for understanding when RA will provide one-copy serializable access. For example, we can apply this theorem to our use cases from Section 2. In the case of multientity update and read, if clients issue read-only and write-only transactions that obey the RSIW property, their result sets will be one-copy serializable. The RSIW property holds for

equality-based lookup of single records from an index (e.g., fetch from the index and subsequently fetch the corresponding base tuple, each of which was written in the same transaction or was autogenerated upon insertion of the tuple into the base relation). However, the RSIW property does not hold in the event of multituple reads, leading to less intuitive behavior. Specifically, if two different clients trigger two separate updates to an index entry, some clients may observe one update but not the other, and other clients may observe the opposite behavior. In this case, the RAMP protocols still provide a snapshot view of the database according to the index(es)—that is, clients will never observe base data that is inconsistent with the index entries—but nevertheless surface nonserializable database states. For example, if two transactions each update items x and y to value 2, then readers of the index entry for 2 may observe $x = 2$ and $y = 2$ but not both at the same time; this is because two transactions updating two separate items do not have RSIW. Finally, for more general materialized view accesses, point queries and bulk insertions may also have RSIW.

As discussed in Section 2, in the case of indexes and views, it is helpful to view each physical data structure (e.g., a CRDT [Shapiro et al. 2011] used to represent an index entry) as a *collection* of versions. The RSIW property applies only if clients make modifications to the entire collection at once (e.g., as in a `DELETE CASCADE` operation); otherwise, a client may read from multiple transaction item write sets, violating RSIW.

Coupled with an appropriate algorithm ensuring RA isolation, we can ensure one-copy serializable isolation. This addresses a long-standing concern with our work: why is RA somehow "natural" for these use cases (but not necessarily all use cases)? We have encountered applications that do not require one-copy serializable access—such as the mailbox unread message maintenance from Section 2 and, in some cases, index maintenance for non-read-modify-write workloads—and therefore may safely violate RSIW but believe it is a handy principle (or, at the least, rule of thumb) for reasoning about applications of RA isolation and the RAMP protocols.

Finally, the RSIW property is only a *sufficient* condition for one-copy serializable behavior under RA isolation. There are several alternative sufficient conditions to consider. As a natural extension, while RSIW only pertains to pairs of read-only and write-only transactions, one might consider allowing readers to observe multiple write transactions. For example, consider the following history:

$$\begin{aligned} T_1 &\;\; w(x_1); w(y_1), \\ T_2 &\;\; w(u_2); w(z_2), \\ T_3 &\;\; r(x_1); r(z_2). \end{aligned} \tag{6}$$

History (6) is valid under RA and is also one-copy serializable but does not satisfy RSIW: T_3 reads from *two* transactions' write sets. However, consider the following history:

$$\begin{aligned} T_1 &: \; w(x_1); w(y_1), \\ T_2 &: \; w(u_2); w(z_2), \\ T_3 &: \; r(x_1); r(z_\perp), \\ T_4 &: \; r(x_\perp); r(z_2). \end{aligned} \tag{7}$$

History (7) is valid under RA, consists only of read-only and write-only transactions, yet is no longer one-copy serializable. T_3 observes a prefix beginning with $T_\perp; T_1$ while T_4 observes a prefix beginning with $T_\perp; T_2$.

Thus, while there may indeed be useful criteria beyond the RSIW property that we might consider as a basis for one-copy serializable execution under RA, we have observed RSIW to be the most intuitive and useful thus far. One clear criteria is to

search for schedules or restrictions under RA with an acyclic DSG (from Appendix A). The reason why RSIW is so simple for read-only and write-only transactions is that each read-only transaction only reads from one other transaction and does not induce any additional antidependencies. Combining reads and writes complicates reasoning about the acyclicity of the graph.

This exercise touches upon an important lesson in the design and use of weakly isolated systems: by restricting the set of operations accessible to a user (e.g., RSIW read-only and write-only transactions), one can often achieve more scalable implementations (e.g., using weaker semantics) *without* necessarily violating existing abstractions (e.g., one-copy serializable isolation). While prior work often focuses on restricting only operations (e.g., to read-only or write-only transactions [Lloyd et al. 2013; Bailis et al. 2014c; Agrawal and Krishnaswamy 1991], or stored procedures [Thomson et al. 2012; Kallman et al. 2008; Jones et al. 2010], or single-site transactions [Baker et al. 2011]) or only semantics (e.g., weak isolation guarantees [Bailis et al. 2014a; Lloyd et al. 2013; Bailis et al. 2012]), we see considerable promise in better understanding the intersection between and combinations of the two. This is often subtle and almost always challenging, but the results—as we found here—may be surprising.

3.5. 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 [Attiya and Welch 2004].

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 [Birman et al. 2009; Bronson et al. 2013; Chang et al. 2006; Cooper et al. 2008; DeCandia et al. 2007; Helland 2007; Hull 2013; Qiao et al. 2013; Weil 2011]. Our goal in this article is to provide robust and scalable transactional functionality, and, so we first define criteria for "scalability":

Coordination-free execution 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 literature, coordination-free execution for replicated transactions is also called *transactional availability* [Bailis et al. 2014a]. Note that "strong" isolation models like serializability and Snapshot Isolation require coordination and thus limit scalability. Locking is an example of a non-coordination-free implementation mechanism.

Many applications can limit their data accesses to a single partition via explicit data modeling [Das et al. 2010; Qiao et al. 2013; Baker et al. 2011; Helland 2007] or planning [Curino et al. 2010; Pavlo et al. 2012]. However, this is not always possible. In the case of secondary indexing, there is a 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 [Weil 2011], perfect partitioning of data accesses is near impossible. Accordingly:

Partition independence ensures that, in order to execute a transaction, a client only contacts partitions for data items that its transaction directly accesses. 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 literature, partition independence for replicated data is also called *replica availability* [Bailis et al. 2014a] or *genuine partial replication* [Schiper et al. 2010]. Using a centralized validator or scheduler for transactions is an example of a non-partition-independent implementation mechanism.

In addition to the previous 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. We will attempt to minimize the *metadata*—that is, data that the transaction did not itself write but which is required for correct execution. Our algorithms will provide constant-factor metadata overhead (`RAMP-S`, `RAMP-H`) or else overhead linear in transaction size (but independent of data size; `RAMP-F`). As an example of a solution using prohibitive amounts of metadata, 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 7).

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 Appendix B, 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 waiting 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).

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 (lines 20–21) 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, 6–8).

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). The reader calculates a mapping from each item i to the highest-timestamped version of i that appears in the metadata of any version (of i or of any other item) in the 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 2. 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 $y = \emptyset$, 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 $1 > \emptyset$) 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 two 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 $lastCommit[i] = 2$, 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 metadata size linear in write set size but provides best-case one RTT for reads, RAMP-Small (RAMP-S) uses constant metadata but always requires two

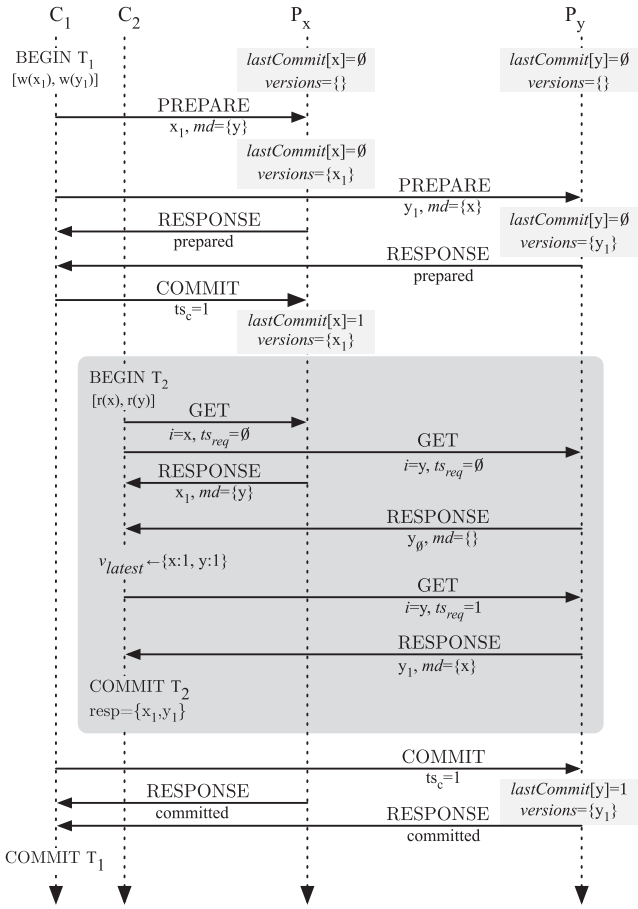


Fig. 2. Space-time diagram for RAMP-F execution for two transactions T_1 and T_2 performed by clients C_1 and C_2 on partitions P_x and P_y. Lightly shaded boxes represent current partition state (*lastCommit* and *versions*), while the single darker box encapsulates all messages exchanged during C_2's execution of transaction T_2. 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. We depict a compressed version of metadata, where each item does not appear in its list of writes (e.g., P_x sees $\{y\}$ only and not $\{x, y\}$).

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). Then 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.

By Example. In Figure 3, under RAMP-S, P_x and P_y, respectively, return the sets $\{1\}$ and $\{\emptyset\}$ in response to T_2's first round of reads. T_2 would subsequently send the set $\{1, \emptyset\}$ to both P_x and P_y, which would return x_1 and y_1.

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

ALGORITHM 1: RAMP-Fast

Server-side Data Structures

1: $versions$: set of versions $\langle item, value$, timestamp ts_v, metadata $md\rangle$
2: $lastCommit[i]$: 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: $\forall i \in I_{ts}, lastCommit[i] \leftarrow \max(lastCommit[i], ts_c)$

9: **procedure** GET(i : item, ts_{req} : timestamp)
10: **if** $ts_{req} = \emptyset$ **then**
11: **return** $v \in versions : v.item = i \wedge v.ts_v = lastCommit[item]$
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: $w \leftarrow \langle item = i, value = v, ts_v = ts_{tx}, md = (I_{tx} - \{i\})\rangle$
19: invoke PREPARE(w) 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

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, $lastCommit[i]$, suffices in line 3).

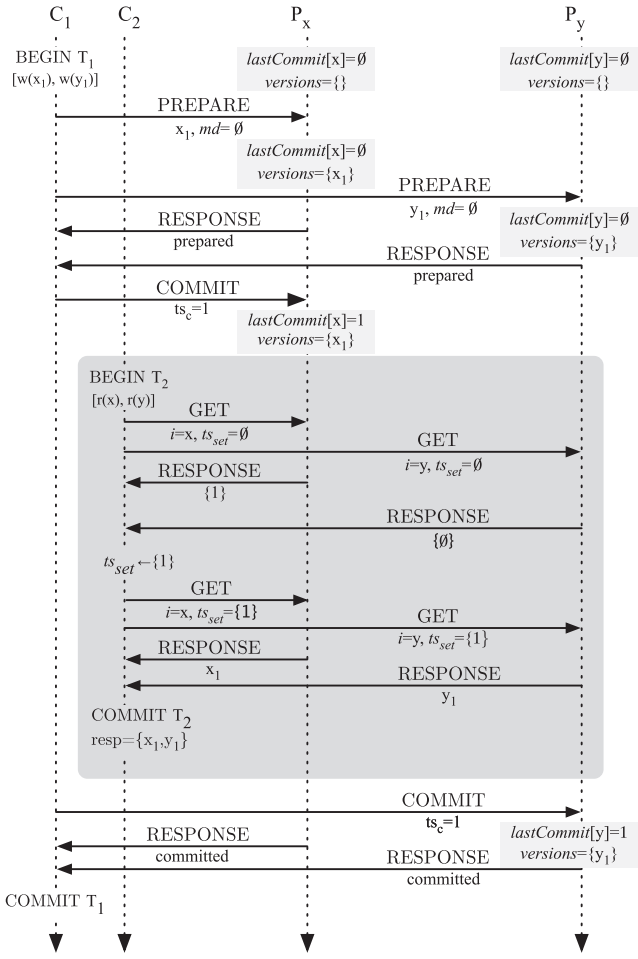


Fig. 3. Space-time diagram for RAMP-S execution for two transactions T_1 and T_2 performed by clients C_1 and C_2 on partitions P_x and P_y. Lightly shaded boxes represent current partition state (*lastCommit* and *versions*), while the single darker box encapsulates all messages exchanged during C_2's execution of transaction T_2. T_2 first fetches the highest committed timestamp from each partition, then fetches the corresponding version. In this depiction, partitions only return timestamps instead of actual versions in response to first-round reads.

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), RAMP-H writers store a Bloom filter [Bloom 1970] 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 11).

By Example. In Figure 2, under RAMP-H, x_1 would contain a Bloom filter with positives for x and y and $y_\emptyset$ would contain an empty Bloom filter. T_2 would check for the presence

ALGORITHM 2: RAMP-Small

Server-side Data Structures
same as in 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 = lastCommit[i]
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
```

of y in x_1's Bloom filter (since x_1's version is 1 and $1 > \emptyset$) 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 I): 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. 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.

Table I. 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)

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	$1+\epsilon$	2	Bloom filter	stamp/item

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 $\langle item, value \rangle$)
    same as RAMP-F PUT_ALL but instantiate *md* on line 18
    with a Bloom filter containing I_{tx}
2: **procedure** GET_ALL(I : set of items)
3: $ret \leftarrow \{\}$
4: **parallel-for** $i \in I$
5: $ret[i] \leftarrow \text{GET}(i, \emptyset)$
6: $v_{fetch} \leftarrow \{\}$
7: **for** version $v \in ret$ **do**
8: **for** version $v' \in ret : v' \neq v$ **do**
9: **if** $v.ts_v > v'.ts_v \wedge v.md.lookup(v'.item) \rightarrow True$ **then**
10: $v_{fetch}[v'.item].\text{add}(v.ts_v)$
11: **parallel-for** item $i \in v_{fetch}$
12: $ret[i] \leftarrow \text{GET}(i, v_{fetch}[i])$ **if** $\text{GET}(i, v_{fetch}[i]) \neq \perp$
13: **return** ret

RAMP algorithms are scalable because clients only contact partitions directly accessed by their transactions (partition independence), and clients cannot stall one another (are coordination-free). 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); however, the reader may have to contact the servers twice in order to fetch any versions that were missing from its first set of reads.

4.5. Additional Details

In this section, we discuss relevant implementation details.

Multiversioning and Garbage Collection. RAMP transactions rely on multiversioning to allow readers to access versions that have not yet committed and/or have been overwritten. In our pseudocode, we have presented an implementation based on multiversioned storage; in practice, multiversioning 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 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 $v < lastCommit[i]$), 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 [Lloyd et al. 2011, 2013]. 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. Under RAMP-F, if a second-round read request arrives at a server and the server does not have that version due to garbage collection, it can safely ignore the request or signal failure.

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 predeclare the data items they wish to read, then the client can execute a GET_ALL transaction at the start of transaction execution to prefetch all items; subsequent accesses to those items can be served from this prefetched 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 predeclared read/write sets [Curino et al. 2010; Pavlo et al. 2012; Thomson et al. 2012] and write buffering [Corbett et al. 2012; Shute et al. 2013], 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 Universally Unique Identifiers (UUIDs) and by hashing transaction contents.

Overwrites. In our algorithms, 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, $lastCommit[i]$ contains an abstract data type (e.g., set of versions) that can be updated with a *merge* operation [DeCandia et al. 2007; Terry et al. 1994] (instead of *update If Greater*) upon commit. This treats each committed record as a set of versions, requiring additional metadata (that can be GCed as in Section 4.8).

4.6. Distribution and Fault Tolerance

RAMP transactions operate in a distributed setting, which poses challenges due to latency, partial failure, and network partitions. Under coordination-free execution,

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. In this section, we address incident concerns. First, replication can be used to increase the number of servers hosting a partition, thus increasing availability. Second, we describe the RAMP protocol behavior when clients are unable to contact servers.

Replication. RAMP protocols can benefit from a variety of mechanisms including traditional database master-slave replication with failover, quorum-based protocols, and state machine replication, which increase the number of physical servers that host a given data item [Bernstein et al. 1987]. To improve durability, RAMP clients can wait until the effects of their operations (e.g., modifications to *versions* and *lastCommit*) are persisted 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. Availability can also benefit in many protocols, such as quorum replication. We discuss more advanced replication techniques in Section 6.1.

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 [Bernstein et al. 1987]. However, under coordination-free execution, 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 Bernstein et al. [1987]. 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. Under CTP, if a client blocks mid-COMMIT, the servers will ensure that the writes will eventually COMMIT and therefore become visible on all partitions. A client recovering from a failure can read from the servers to determine if they unblocked its write.

CTP 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 [Gray and Lamport 2006]), we have found CTP to be both lightweight and effective. We evaluate CTP in Section 5.3.

4.7. Additional Semantics

While our RAMP transactions provide RA isolation, they also provide a number of additional useful guarantees. With linearizable servers, once a user's operation completes, all other users will observe its effects (regular register semantics, applied at the transaction level); this provides a notion of real-time recency. This also ensures that each user's operations are visible in the order in which they are completed. Our RAMP implementations provide a variant of PRAM consistency, where for each item, each user's

writes are serialized [Lipton and Sandberg 1988] (i.e., "session" ordering [Daudjee and Salem 2004]). 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 [Cooper et al. 2008]. However, Parallel Random Access Memory (PRAM) does not respect the *happens-before* relation [Lamport 1978] across users (or missing dependencies, as discussed in Section 3.3). If Sam reads Mary's comment and replies to it, other users may read Sam's comment without Mary's comment. We further discuss this issue in Section 6.3.

4.8. Further Optimizations

RAMP algorithms also allow several possible optimizations:

Faster Commit Detection. If a server returns a version in response to a GET request, then the transaction that created the version must have issued a COMMIT on at least one server. In this case, the server can safely mark the version as committed and update *lastCommit*. This means that the transaction commit will be reflected in any subsequent GET requests that read from *lastCommit* for this item—even though the COMMIT message from the client may yet be delayed. The net effect is that the later GET requests may not have to issue second-round reads to fetch the versions that otherwise would not have been marked 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, nontimestamp metadata for T's writes stored in RAMP-F and RAMP-H (write sets and Bloom filters) can 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 Section 4.7), 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 [Gray and Lamport 2006], the servers can exchange PREPARE acknowledgments with one another and decide to COMMIT autonomously. This optimization is safe because multiple PREPARE phases can safely overlap. We leverage a similar observation in Section 6.1.

5. EXPERIMENTAL EVALUATION

We proceed to experimentally demonstrate RAMP transaction scalability as compared to existing transactional and nontransactional 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 coordination-free 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, multiversioned,

main-memory database prototype. Our prototype is in Java and employs a custom Remote Procedure Call (RPC) system with Kryo 2.20 for serialization. Servers are arranged as a distributed hash table [Stoica et al. 2001] with partition placement determined by random hashing of keys to servers. As in stores like Dynamo [DeCandia et al. 2007], 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 5s 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. We chose not to employ metadata garbage collection and one-phase writes 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 [Gray et al. 1976]: 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 (`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 [Lloyd et al. 2013] (denoted `E-PCI`; Section 7). Writes proceed via prepare and commit rounds, but any reads that arrive at a partition and while a write transaction to the same item is pending 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—in this case—is still coordination-free. 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 [Cooper et al. 2010] and deploy variably sized sets of servers on public cloud infrastructure. We employ `cr1.8xlarge` instances on Amazon EC2, each containing 32 virtual CPU cores (Intel Xeon E5-2670) and 244GB RAM. By default, we deploy five partitions on five servers. We group sets of reads and sets of writes into read-only and write-only transactions (default size: four 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, Bronson et al. [2013], Lloyd et al. [2013], and Weil [2011]; e.g., Tao's 500 to 1 reads-to-writes [Bronson et al. 2013; Lloyd et al. 2013], Espresso's 1000 to 1 Mailbox application [Qiao et al. 2013], and Spanner's 3396 to 1 advertising application [Corbett et al. 2012]).

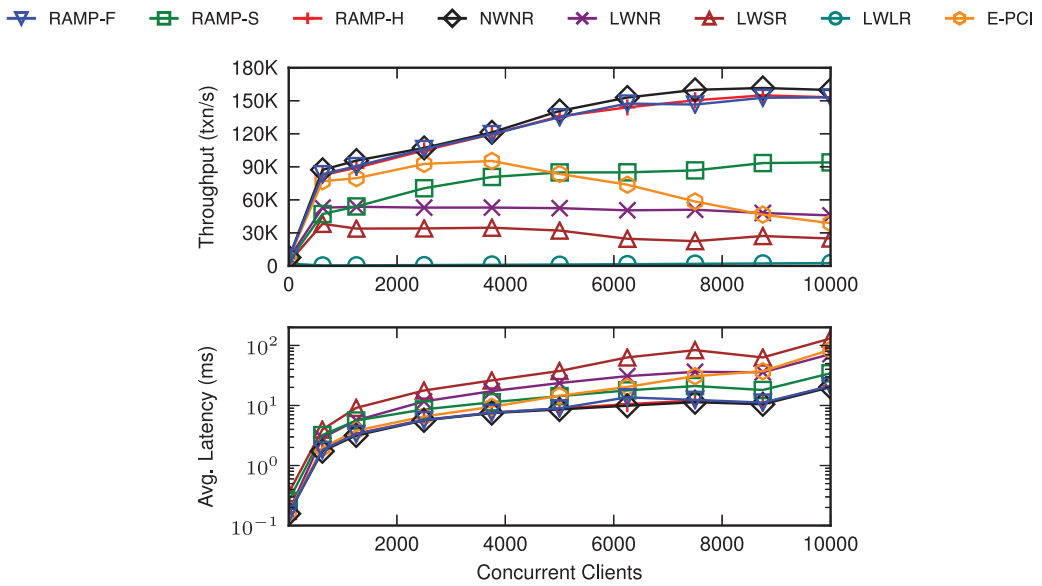


Fig. 4. Throughput and latency under varying client load. We omit latencies for LWLR, which peaked at over 1.5s.

By default, use 5000 YCSB clients distributed across five separate EC2 instances. As in stock YCSB, each client makes a sequence of synchronous requests to the database. When we later vary the number of clients, we keep the number of servers hosting the clients fixed. 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 size of L writes, 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 size, value size, database size, and number of servers and report the average of three 60s trials.

Reproducibility. All source code used in experiments is available at https://www.github.com/pbailis/.

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 4). 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 with 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 10ms at 5000 clients; RAMP-S less than 53ms, 14.3ms at 5000 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. Under YCSB's Zipfian workload, there is a high probability that the several "hot" keys in the workload have a pending write, requiring a E-PCI commit check. The number of in-flight writes further increases with load, increasing the number of E-PCI commit checks. This in turn decreases throughput, and with 10,000 concurrent clients, E-PCI performs so many commit checks per read that it underperforms the LWNR lock-based scheme. Under this configuration, more than 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. Meanwhile, multipartition locking is expensive [Pavlo et al. 2012]: 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% and peaks at 3412 transactions per second.

We subsequently varied several other workload parameters, which we briefly discuss in the following and plot in Figure 5.

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 RTTs) 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 Size. Increased transaction sizes (i.e., number of operations) have variable impact on the relative performance of RAMP algorithms. Coordination-free execution ensures long-running transactions are not penalized, but with longer transactions, metadata overheads increase. RAMP-F relative throughput decreases due to additional metadata (linear in transaction size) and RAMP-H relative performance also decreases as its Bloom filters saturate. (However, YCSB's Zipfian-distributed access patterns result in a nonlinear relationship between size and throughput.) As discussed earlier, 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 sizes (e.g., 1024 bit filters should lower the false positive rate for transactions of size 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 move from 1B to 100KB.

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%

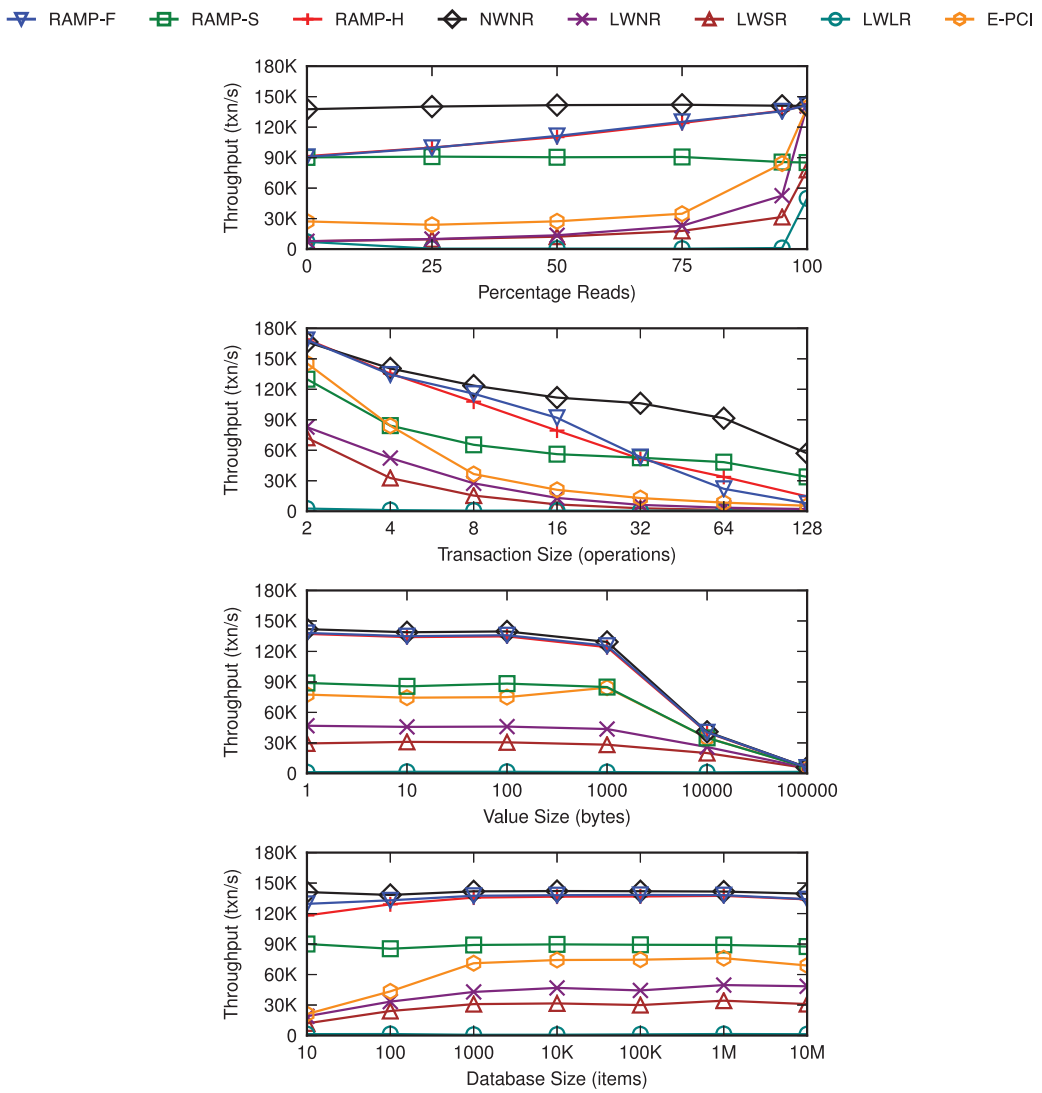


Fig. 5. Algorithm performance across varying workload conditions. RAMP-F and RAMP-H exhibit similar performance to NWNR baseline, while RAMP-S's two RTT reads incur a greater performance penalty across almost all configurations. RAMP transactions consistently outperform RA isolated alternatives.

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 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. E-PCI triggers many commit checks but 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 CTP 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 following table 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 5 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 previous 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 10GbE connections between instances; so, past 20 servers, servers communicated over a degraded network. At 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. To avoid bottlenecks on the client, we deploy as many instances to host YCSB clients as we do to host prototype servers. However, as shown in Figure 6, each RAMP algorithm scales linearly. In expectation, at 100 servers, almost all transactions span multiple servers: all but one in 100M transactions is a multipartition operation, highlighting the importance of partition independence. With 100 servers, 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.

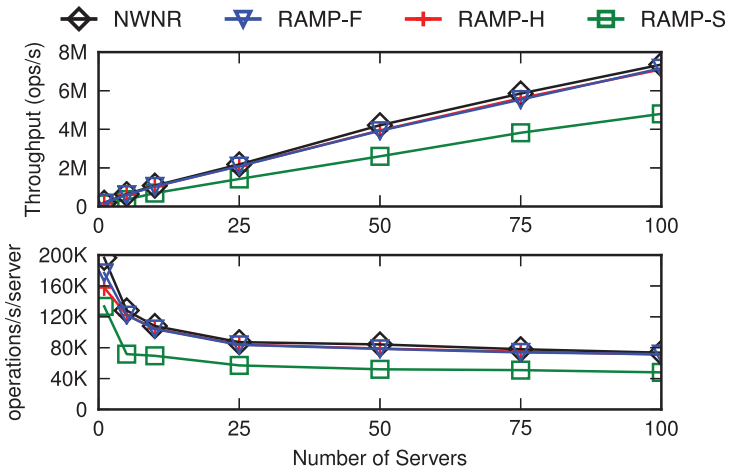


Fig. 6. RAMP transactions scale linearly to over 7 million operations/s with comparable performance to NWNR baseline.

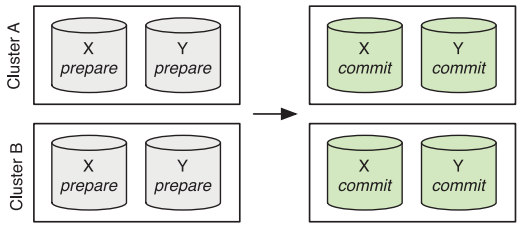
6. APPLYING AND MODIFYING THE RAMP PROTOCOLS

In this section, we discuss modifications to RAMP to enable multidatacenter and efficient quorum replication as well as causally consistent operation. Our goals here are twofold. First, we believe this section will be beneficial to systems implementers integrating RAMP protocols into databases such as Cassandra [Lakshman and Malik 2008] that support wide-area and quorum-replicated deployments. Second, we believe this material will be useful to readers who are familiar with existing and recent work on both multidatacenter and causally consistent replication. Namely, RAMP is compatible with many of these replication scenarios, and in some cases, enables new optimizations.

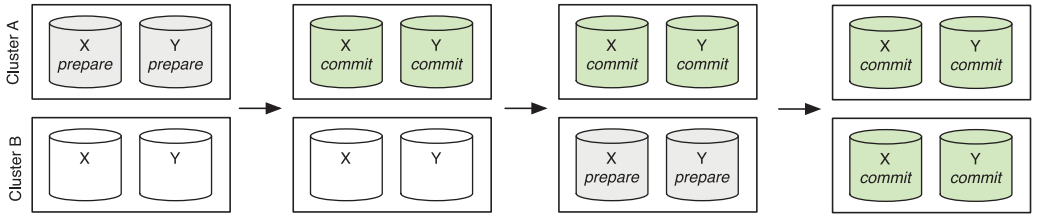
6.1. Multidatacenter RAMP

The RAMP algorithms presented in this work have assumed linearizable server operation. Hence, if RAMP is used in a system where data items are replicated, then a linearizable replication mechanism must be used, such as a primary-backup or other replicated state machine approach. While this has simplified our discussion and results in reasonable performance in many environments, the cost of linearizability is often expensive, particularly in geo-replicated environments where latency is lower-bounded by the speed of light [Bailis et al. 2014a; Abadi 2012]. While the RAMP algorithms' lack of coordination mitigates throughput penalties due to, for example, stalls during contended multipartition access, actually accessing the partitions themselves may take time and increase the latency of individual operations. Moreover, in the event of partial failures, it is often beneficial to provide greater availability guarantees.

In this section, we discuss strategies for lowering the latency and improving the availability of operations. Our primary target in this setting is a multidatacenter, geo-replicated context, where servers are located in separate clusters in possibly geographically remote regions. This setting has received considerable attention in recent research and, increasingly, in some of the largest production data management systems [Sovran et al. 2011; Lloyd et al. 2011, 2013; Corbett et al. 2012]. The actual porting of concurrency control algorithms to this context is not terribly difficult, but any inefficiencies due to synchronization and coordination are magnified in this setting, making it an ideal candidate for practical study. Thus, we couch our discussion in the context of fully replicated clusters (i.e., groups of replicas of each partition).



(a) High availability multicluster RAMP



(b) Sticky multicluster RAMP

Fig. 7. Control flow for operations under multidatacenter RAMP strategies with client in Cluster A writing to partitions X and Y. In the high availability RAMP strategy (Figure 7(a)), a write must be prepared on $F + 1$ servers (here, $F = 3$) before is committed. In the sticky RAMP strategy, a write can be prepared and committed within a single datacenter and asynchronously propagated to other datacenters, where it is subsequently prepared and committed (Figure 7(b)). The sticky strategy requires that clients maintain affinity with a single cluster in order to guarantee available and correctly isolated behavior.

The key challenge in achieving higher availability and lower latency in RAMP is ensuring that partially committed writes can still be completed. In the standard RAMP algorithms, this is accomplished by waiting to commit until after all partitions have prepared. Yet, in a replicated context, this waiting is potentially expensive; over wide-area networks, this can take hundreds of milliseconds. There are two straightforward ways to circumvent these overheads: deferring the commit operation and maintaining stickiness.

Prepare-F HA RAMP. The first strategy is easier to understand but perhaps less practical. A client specifies a minimum durability for its write operations, measured in terms of number of failures it wishes to survive, F. When writing, the client issues a prepare request to all clusters and waits until it receives a successful response from $F + 1$ servers. This ensures that the client's write is durable, and the client knows its intent has been logged on at least $F + 1$ servers. The client transaction subsequently returns success (Figure 7(a)). Once all servers have received the prepare request (detectable via either server-server communication as in the CTP protocol or via an asynchronous callback on the client), the servers can begin to commit the client's writes autonomously. This preserves RA isolation—but at a cost. Namely, there is no guarantee of *visibility* of writes: a client is not guaranteed to read its own writes. Moreover, if a single server is offline, the servers will not begin the commit step, and clients will not observe the effects of the prepared transactions for an indefinite period of time. By ensuring availability of writes (i.e., clients return early), we have sacrificed visibility in the form of ensuring that writes are accessible to readers. Thus, clients will not enjoy session guarantees [Terry et al. 1994] such as Read Your Writes. Given the importance of these session guarantees for many of the industrial users we have encountered (e.g., see Facebook's TAO geo-replication [Bronson et al. 2013]), we currently do not favor this approach.

Sticky HA RAMP. The second strategy is to ensure a degree of stickiness, or affinity, between clients and servers within a datacenter [Bailis et al. 2014a]. Each client is assigned its own datacenter. Instead of having a client issue its writes to the entire database replica set, the client can instead issue its prepare and commit operations to its assigned datacenter (or local replica group) and subsequently forward the writes to be prepared and committed autonomously in separate clusters (Figure 7(b)). That is, once a writer has performed the appropriate RAMP protocol in its assigned datacenter, it can return. In an N-datacenter deployment, each full write protocol is performed N separate times—once per datacenter. If the same timestamp is assigned to each write, the end state of each datacenter will be equivalent. As long as clients remain connected to the *same* datacenter (i.e., is "sticky" with respect to its database connections), it will read its writes.

The total number of prepare and commit operations is the same as in the first strategy, but the commit point is staggered—each cluster reaches a commit point independently, at different times. Moreover, clusters operate independently, so throughput is not improved—only latency—because each cluster must replay every other cluster's writes [Bailis et al. 2012]. This is the basic strategy espoused by traditional log shipping approaches [Ladin et al. 1992] as well as more recent proposals such as the COPS [Lloyd et al. 2011] and Eiger [Lloyd et al. 2013] systems.

However, this stickiness has an often-neglected penalty: a client can no longer connect to arbitrary servers and expect to read its own writes. If a server is down in a client's local datacenter, the client must—in the worst case—locate an entire other replica set to which the client can connect. This negatively affects availability: the *Prepare-F* strategy can utilize all servers at once, but the sticky strategy requires clients to maintain affinity for availability. In cases when this "sticky availability" [Bailis et al. 2014a] is acceptable (e.g., each datacenter contains a set of application servers that issue the RAMP protocols against another datacenter-local set of storage servers), this may be a reasonable compromise.

6.2. Quorum-Replicated RAMP Operation

While RAMP *Prepare-F* and *Sticky HA* are best suited for multidatacenter deployments, in quorum-replicated systems such as Dynamo and Cassandra [Lakshman and Malik 2008; DeCandia et al. 2007], there are several optimizations that can be used to further improve availability, even within a single datacenter.

Our key observation here is that, to guarantee maximum two-round-trips for reads, only PREPARE and second-round GET requests need to intersect on a given set of replicas. Recall that second-round GET requests are issued in order to "repair" any fractured reads from the first round of read results. In the event of these fractured reads, a reader *must* have access to versions corresponding to fractured reads that have been prepared but were not committed at the time of the first-round read. However, assembling the first round of committed versions can run under partial (i.e., nonintersecting) quorum operation [Malkhi et al. 2001] with respect to commit messages.

This means that COMMIT and first-round GET operations can proceed on effectively any server in a set of replicas, enabling two key optimizations. In these optimizations, we assume that readers issue second-round read requests and writers issue PREPARE operations using a quorum system [Naor and Wool 1998] of replicas (e.g., majority quorums).

First, first-round read requests can be served from any replica of a given item. Then, if a client detects a race (in `RAMP-F` or `RAMP-H`), it can issue the optional second round of requests to a quorum of servers. `RAMP-S` will always issue the second round of requests. This optimization improves the latency of the first round of reads and also enables

speculative retry [Dean and Barroso 2013]. It also decreases the load on the servers and increases availability for first-round read operations.

Second, commit operations can be performed on any replica of a given item. Similar to the optimization proposed in *Prepare-F* RAMP, servers can propagate commit messages between themselves asynchronously, possibly piggybacking on antientropy messages as in systems like Cassandra and Dynamo. This optimization improves the latency of commit. However, because all servers must commit the transaction eventually, it does not necessarily decrease the load on servers.

To quantify the potential latency improvements achievable using these optimizations, we draw on latency distributions from a recent study of Dynamo-style operation [Bailis et al. 2014d]. According to latency data from a Dynamo-style quorum-replicated database running on spinning disks at LinkedIn, moving from waiting for two replicas of three to respond ($N = 3$, $R = 1$) to waiting for one replica of three to respond ($N = 3$, $R = 1$) to a write request decreased latency from 21.0ms to 11.0ms at the 99.9th percentile; 1.63ms to 0.66ms for reads. For a similar database at Yammer, the gains for writes are 427ms to 10.8ms and the gains for reads are 32.6ms to 5.6ms—an even more impressive gain. Over a wide-area network with latency of 75ms, the gains are as much as 148ms. Thus, in practice, these simple optimizations may prove worthwhile.

6.3. RAMP, Transitive Dependencies, and Causal Consistency

In Section 3.3, we discussed how RA isolation does not enforce transitive read-write dependencies across transactions. For example, if T_a read-depends on T_b (i.e., T_a reads a version that T_b created), another transaction T_c might read-depend on T_a (i.e., T_c reads a version that T_a created) but antidepend on T_b (i.e., T_b overwrites a version that T_a read). In this section, we discuss why we made this design decision as well as alternatives for enforcing dependencies and their costs.

The primary challenges in enforcing transitive dependencies come in limiting metadata while preserving availability and partition independence. In the extreme, if we limited ourselves to serial access to database state, we could easily preserve information about dependencies using a single scalar: any transactions would observe versions with lower scalar values, similar to classic serializable multiversion concurrency control. However, if we wish to preserve available and coordination-free operation (and therefore concurrent creation of versions), then we must admit a partial ordering of versions. To avoid fractured reads as in RA isolation while preserving dependency information, we either need to find a way to capture this partial order or otherwise limit the degree of availability in the system.

Full Causality Tracking. The former approach—tracking "cuts" in a system with partially ordered events—is well studied. As a first approximation, we can consider the problem of capturing RA with dependency tracking as an instance of capturing causality in a distributed system, with each event corresponding to a transaction commit and dependencies due to reads (i.e., a causal memory with atomically visible, multiregister reads). In line with this approximation, we could replace each timestamp in the RAMP algorithms with a suitable mechanism for tracking causality; for example, instead of storing a scalar timestamp, we could store a vector clock, with one entry per client in the system. Subsequently, clients could maintain a vector clock containing the highest-committed writes they had seen, and upon reading from servers, ensure that the server commits any writes that happen before the client's current vector. Thus, we can use vector clocks to track dependencies across transactions.

The problem with the preceding approach is in the size of the metadata required. Primarily, with N concurrent clients, each vector will require $O(N)$ space, which is

potentially prohibitive in practice. Moreover, the distributed systems literature strongly suggests that, with N concurrent clients, $O(N)$ space is *required* to capture full causal lineage as previously [Charron-Bost 1991]. Thus, while using vector clocks to enforce transitive dependencies is a correct approach, it incurs considerable overheads that we do not wish to pay and have yet to be proven viable at scale in practical settings [Bailis et al. 2012].[2]

The latter approach—limiting availability—is also viable, at the cost of undercutting our scalability goals from Section 3.5.

Bound Writer Concurrency. One simple approach—as we hinted earlier—is to limit the concurrency of writing clients: we can bound the overhead of vector clocks to an arbitrarily small amount by limiting the amount of concurrency in the system. For example, if we allow five clients to perform writes at a given time, we only need a vector of size five. This requires coordination between writers (but not readers). As Section 5.2 demonstrated, RAMP transaction performance degrades gracefully under write contention; under the decreased concurrency strategy, performance would effectively hit a cliff. Latency would increase due to queuing delays and write contention, and for a workload like YCSB with a fixed proportion of read to write operations, throughput would be limited. Specifically, for a workload with p writers ($p = 0.05$ in our default configuration), if W writers were permitted at a given time, the effective number of active YCSB clients in the system would become $\frac{W}{p}$. Despite these limits, this is perhaps the most viable solution we have encountered and, moreover, does not affect read performance under read-heavy workloads.

Sacrificing Partition Independence. Another approach to improving availability is to sacrifice partition independence. As we discuss and evaluate in Sections 5.1 and 5.2, it is possible to preserve transaction dependencies by electing special coordinator servers as points of rendezvous for concurrently executing transactions. If extended to a non-partition-independent context, the RAMP protocols begin to more closely resemble traditional multiversion concurrency control solutions, in particular Chan and Gray [1985]. More recently, the 2PC-PCI mechanism [Lloyd et al. 2013] we evaluated is an elegant means of achieving this behavior if partition independence is unimportant. Nevertheless, as our experimental evaluation shows, sacrificing this partition independence can be costly under some workloads.

Sacrificing Causality. A final approach to limiting the overhead of dependency tracking is to limit the number of dependencies to track. Several prior systems have used limited forms of causality, for example, application-supplied dependency information [Bailis et al. 2013; Ladin et al. 1992], as a basis for dependency tracking. In this strategy, applications inform the system about what versions should precede a given write; in Bailis et al. [2012], we show that, for many modern web applications, these histories can be rather small (often one item, with a power-law distribution over sizes). In this case, it is possible that we could encode the causal history in its entirety along with each write, or exploit otherwise latent information within the data such as comment `reply-to` fields to mine this data automatically. This strategy breaks the current RAMP API, so we do not consider it further. However, when possible, it is the only known strategy for circumventing the $O(N)$ upper bound on dependency tracking in causally consistent storage systems ensuring availability of both readers and writers.

[2]Another alternative that uses additional metadata is the strawman from Section 3.5, in which clients send all of the writes in their transaction to all of the partitions responsible for at least one write in the transaction. This uses even more metadata than the vector-based approach.

Experiences with Systems Operators. While causal consistency provides a number of useful guarantees, in practice, we perceive a lack of interest in maintaining full causal consistency; database operators and users are often unwilling to pay the metadata and implementation costs of full causality tracking. As we have seen in Section 2, many of these real-world operators exhibit an aversion to synchronization at scale, so maintaining availability is paramount to either their software offerings or business operation. In fact, we have found coordination-free execution and partition independence to be valuable selling points for the RAMP algorithms presented in this work. Instead, we have found many users instead favor guarantees such as Read Your Writes (provided by the RAMP algorithms) rather than full dependency tracking, opting for variants of explicit causality (e.g., via foreign key constraints or explicit dependencies) or restricted, per-item causality tracking (e.g., version vectors [DeCandia et al. 2007]). Despite this mild pessimism, we view further reduction of causality overhead to be an interesting area for future work—including a more conclusive answer to the availability-metadata trade-off surfaced by Charron-Bost [1991].

7. RELATED WORK

Replicated databases offer a broad spectrum of isolation guarantees at varying costs to performance and availability [Bernstein et al. 1987]:

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 [Bernstein et al. 1987; Agrawal and Krishnaswamy 1991], multiversion concurrency control (e.g., Phatak and Badrinath [1999]), locking (e.g., Llirbat et al. [1997]), and optimistic concurrency control [Shute et al. 2013]. 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 [Bailis et al. 2014a; Davidson et al. 1985]. Many designs [Kallman et al. 2008; Das et al. 2010] exploit cheap serializability within a single partition but face scalability challenges for distributed operations. Recent industrial efforts like F1 [Shute et al. 2013] and Spanner [Corbett et al. 2012] have improved performance via aggressive hardware advances, but their reported throughput is still limited to 20 and 250 writes per item per second. Multipartition serializable transactions are expensive and, especially under adverse conditions, are likely to remain expensive [Curino et al. 2010; Jones et al. 2010; Pavlo et al. 2012].

Weak Isolation. The remainder of the isolation spectrum is more varied. Most real-world databases offer (and often default to) nonserializable isolation models [Mohan 2013; Bailis et al. 2014a]. 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 [Berenson et al. 1995]; by preventing Lost Update, concurrency control mechanisms providing Snapshot Isolation require coordination [Bailis et al. 2014a]. In recent years, many "NoSQL" designs have avoided cross-partition transactions entirely, effectively providing Read Uncommitted isolation in many industrial databases such PNUTS [Cooper et al. 2008], Dynamo [DeCandia et al. 2007], TAO [Bronson et al. 2013], Espresso [Qiao et al. 2013], Rainbird [Weil 2011], and BigTable [Chang et al. 2006]. 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 [Lloyd et al. 2011] 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 [Lloyd et al. 2013] 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 [Das et al. 2010] but with weaker isolation guarantees; each coordinator effectively contains a partitioned completed transaction list from Chan and Gray [1985]. 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 MAV isolation: transactions read from a monotonically advancing view of database state [Bailis et al. 2014a]. 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 Bailis et al. [2014a] is similar to RAMP-F and, 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., nonlinearizable, "AP/EL" [Gilbert and Lynch 2002; Abadi 2012]) replicated setting of Bailis et al. [2014a], albeit with necessary penalties to latency between updates and their visibility.

A portion of this work originally appeared in SIGMOD as Bailis et al. [2014c]. This invited article expands upon that work. It provides an extended discussion of the RA isolation model (Section 3.3), the introduction of the RSIW model (Section 3.4) and proof of serializability for RA isolation under the RSIW property (Appendix A), and applications of the basic RAMP algorithms to multidatacenter deployments (Section 6.1) and causally consistent replication (Section 6.3). Since the publication of Bailis et al. [2014c], we have leveraged the RAMP algorithms in the development of coordination avoiding implementations of transactional workloads including TPC-C [Bailis et al. 2015] and, more recently, in Object Relational Mapping (ORM)-backed workloads. RAMP has also received attention in the developer community. A proposal for RAMP implementation in Cassandra is currently under review, at least one other production NoSQL store has a RAMP implementation under way, and RA isolation was featured in Facebook's Apollo database roadmap as an alternative to strongly consistent operation.

We are unaware of another protocol for partitioned databases that ensures coordination-free execution, partition independence, and at least RA isolation.

8. CONCLUSIONS AND EXTENSIONS

This article described how to achieve atomically visible multipartition transactions without incurring the performance and availability penalties of traditional algorithms. We first identified a new isolation level—RA 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 multiversioning 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 coordination-free 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 applications for which they are appropriate may benefit measurably.

Given our experiences designing and evaluating the RAMP transaction protocols, we believe there are a number of interesting extensions that merit further examination.

First, RAMP metadata effectively encodes a limited form of transaction *intent*: readers and servers are only able to repair fractured reads because the metadata encodes the remainder of the work required to complete the transaction. We believe it would be interesting to generalize this concept to arbitrary program logic: for example, in a model such as lazy transactions [Faleiro et al. 2014] or eventual serializability [Fekete et al. 1996], with transactions expressed as stored procedures, multiple, otherwise conflicting/coordinating clients could instead cooperate in order to complete one anothers' transactions in the event of a failure—without resorting to the use of a centralized master (e.g., for prescheduling or validating transaction execution). This programming model is largely incompatible with traditional interactive transaction execution but is nevertheless exciting to consider as an extension of these protocols.

Second, and more concretely, we see several opportunities to extend RAMP to more specialized use cases. The RAMP protocol family is currently not well suited to large scans and, as we have discussed, does not enforce transitive dependencies across transactions. We view restricting the concurrency of writers (but not readers) to be a useful step forward in this area, with predictable impact on writer performance. This strikes a middle ground between traditional MVCC and the current RAMP protocols.

Finally, as we noted in Section 3.4, efficient transaction processing often focuses on weakening semantics (e.g., weak isolation) or changing the programming model (e.g., stored procedures as previously). As our investigation of the RSIW property demonstrates, there may exist compelling combinations of the two that yield more intuitive, high-performance, or scalable results than examining semantics or programming models in isolation. Addressing this question is especially salient for the many users of weak isolation models in practice today [Bailis et al. 2014a], as it can help one understand when applications require stronger semantics and when, in fact, weak isolation is not simply fast but is also "correct."

APPENDIX

A. RSIW PROOF

To begin, we first show that there exists a well-defined total ordering of write-only transactions in a history that is valid under RA isolation. This will be useful in ordering write transactions in our one-copy equivalent execution.

LEMMA A.1 (WELL-DEFINED TOTAL ORDER ON WRITES). *Given a history H containing read-only and write-only transactions that is valid under RA isolation, $DSG(H)$ does not contain any directed cycles consisting entirely of write-dependency edges.*

PROOF. H is valid under RA isolation and therefore does not exhibit phenomenon *G1c*. Thus, H does not contain any directed cycles consisting entirely of dependency edges. Therefore, H does not contain any directed cycles consisting entirely of write-dependency edges. □

We will also need to place read-only transactions in our history. To do so, we show that, under RA isolation and the RSIW property (i.e., the preconditions of Theorem 3.29), each read-only transaction will only read from one write-only transaction.

LEMMA A.2 (SINGLE READ DEPENDENCY). *Given a history H containing read-only and write-only transactions that obeys the RSIW property and is valid under RA isolation, each node in $DSG(H)$ contains at most one direct read-dependency edge.*

PROOF. Consider a history H containing read-only and write-only transactions that has RSIW and is valid under RA isolation. Write-only transactions have no reads, so they have no read-dependency edges in $DSG(H)$. However, suppose $DSG(H)$ contains a node corresponding to a read-only transaction T_r containing more than one direct read-dependency edge. For two read-dependency edges to exist, T_r must have read versions produced by at least two different write-only transactions; pick any two and call them T_i and T_j, corresponding to read versions x_a and y_d.

If x and y are the same item, then $a < d$ or $d < a$. In either case, T_r exhibits the fractured reads phenomenon and H is not valid under RA isolation, a contradiction.

Therefore, x and y must be distinct items. Because H obeys the RSIW property, T_r must also obey the RSIW property. By the definition of the RSIW property, T_r must have only read items written to by T_i and items also written to by T_j; this implies that T_i and T_j each wrote to items x and y. We can label T_i's write to y as y_b and T_j's write to x as x_c. Per Lemma A.1, T_i's writes to x and y must either both come before or both follow T_j's corresponding writes to x and y in the version order for each of x and y; that is, either both $a < b$ and $c < d$ or both $b < a$ and $d < c$.

If $a < b$ and $c < d$, then T_r exhibits the fractured reads phenomenon: T_r read x_a and y_d but T_j, which wrote y_d also wrote x_b, and $a < b$. If $b < a$ and $d < c$, then T_r again exhibits the fractured reads phenomenon: T_r read x_a and y_d but T_i, which wrote x_a, also wrote y_c, and $d < c$. In either case, H is not valid under RA isolation, a contradiction.

Therefore, each node in $DSG(H)$ must not contain more than one read-dependency edge. □

We now use this ordering on reads and writes to construct a total ordering on transactions in a history:

PROCEDURE 1 (TRANSFORM). *Given a history H containing read-only and write-only transactions that has RSIW and is valid under RA isolation, construct a total ordering O of the transactions in H as follows:*

(1) *Perform a topological sorting in O of each pair of committed write-only transactions in H ordered by the write-dependency edges in $DSG(H)$. That is, for each pair of write-only transactions (T_1, T_2) in H that performed at least one write to the same item, place the transaction that wrote the higher-versioned item later in O. Lemma A.1 ensures such a total ordering exists.*
(2) *For each committed read-only transaction T_r in H, place T_r in O after the write-only transaction T_w whose writes T_r read (i.e., after the write-only transaction that T_r directly read-depends on) but before the next write-only transaction $T_{w'}$ in O (or, if no such transaction exists, at the end of O). By Lemma A.2, each committed read-only transaction read-depends on only one write-only transaction, so this ordering is similarly well defined.*

Return O.

As an example, consider the following history:

$$\begin{aligned} T_1 &\quad w(x_1); w(y_1), \\ T_2 &\quad w(x_2); w(y_2), \\ T_3 &\quad r(x_1); r(y_1), \\ T_4 &\quad\quad r(y_2). \end{aligned} \tag{8}$$

History (8) obeys the RSIW property and is also valid under RA isolation. Applying procedure TRANSFORM to History 8, in the first step, we first order our write-only transactions T_1 and T_2. Both T_1 and T_2 write to x and y, but T_2's writes have a later version number than T_1's, so, according to Step 1 of TRANSFORM, we have $O = T_1; T_2$. Now, in Step 2 of TRANSFORM, we consider the read-only transactions T_3 and T_4. We place T_3 after the transaction that it read from (T_1) and before the next write transaction in the sequence (T_2), yielding $O = T_1; T_3; T_2$. We place T_4 after the transaction that it read from (T_2) and, because there is no later write-only transaction following T_2 in O, place T_4 at the end of O, yielding $O = T_1; T_3; T_2; T_4$. In this case, we observe that, as Theorem 3.29 suggests, it is possible to TRANSFORM an RSIW and RA history into a one-copy serial equivalent and that O is in fact a one-copy serializable execution.

Now we can prove Theorem 3.29. We demonstrate that executing the transactions of H in the order resulting from applying TRANSFORM to H on a single-copy database yields an equivalent history (i.e., read values and final database state) as H. Because O is a total order, H must be one-copy serializable.

PROOF OF THEOREM 3.29. Consider history H containing read-only and write-only transactions that has RSIW and is valid under RA isolation. We begin by applying TRANSFORM to H to produce an ordering O.

We create a new history H_o by considering an empty one-copy database and examining each transaction T_h in O in serial order as follows: if T_h is a write-only transaction, execute a new transaction T_{ow} that writes each version produced by T_h to the one-copy database and commits. If T_h is a read-only transaction, execute a new transaction T_{or} that reads from the same items as T_h and commits. H_o is the result of a serial execution of transactions over a single logical copy of the database; H_o is one-copy serializable.

We now show that H and H_o are equivalent. First, all committed transactions and operations in H also appear in H_o because TRANSFORM operates on all transactions in H and all transactions and their operations (with single-copy operations substituted for multiversion operations) appear in the total order O used to produce H_o. Second, $DSG(H)$ and $DSG(H_o)$ have the same direct read dependencies. This is a straightforward consequence of Step 2 of TRANSFORM: in O, each read-only transaction T_r appears immediately following the write-only transaction T_w upon which T_r read-depends. When the corresponding read transaction is executed against the single-copy database in H_o, the serially preceding write-only transaction will produce the same values that the read transaction read in H. Therefore, H and H_o are equivalent.

Because H_o is one-copy serializable and H_o is equivalent to H, H must also be one-copy serializable. □

We have opted for the preceding proof technique because we believe the TRANSFORM procedure provides clarity into *how* executions satisfying both RA isolation and the RSIW property relate to their serializable counterparts. An alternative and elegant proof approach leverages the work on multiversion serializability theory [Bernstein et al. 1987], which we briefly sketch here. Given a history H that exhibits RA isolation and has RSIW, we show that $MVSG(H)$ is acyclic. By an argument resembling Lemma A.2, the in-degree for read-only transactions in $SG(H)$ (i.e., Adya's direct read dependencies) is one. By an argument resembling Lemma A.1, the edges between write-only transactions in $MVSG(H)$ produced by the first condition of the *MVSG* construction ($x_i \ll x_j$ in the definition of the *MVSG* [Bernstein et al. 1987, p. 152]; that is, Adya's write dependencies) are acyclic. Therefore, any cycles in the *MVSG* must include at least one of the second kind of edges in the $MVSG(H)$ ($x_j \ll x_i$; i.e., Adya's direct antidependencies). But, for such a cycle to exist, a read-only transaction T_r must antidepend on a write-only transaction T_{wi} that in turn write-depends on another

write-only transaction T_{wj} upon which T_r read-depends. Under the RSIW property, T_{wi} and T_{wj} will have written to at least one of the same items, and the presence of a write-dependency cycle will indicate a fractured reads anomaly in T_r.

B. RAMP CORRECTNESS AND INDEPENDENCE

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 say that x is a sibling item to a write y_j if there exists a version x_k that was written in the same transaction as y_j.

Given two versions x_i and y_j, we say that x_i is a *companion version* to y_j if x_i is a transactional sibling of y_j or if the transaction that wrote y_j also wrote x_k and $i > k$. 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 2, the versions returned by T_2's first round of reads ($\{x_1, y_\emptyset\}$) do not comprise a companion set because $y_\emptyset$ has a lower timestamp than x_1's sibling version of y (that is, x_1 has sibling version y_1 and but $\emptyset < 1$ so $y_\emptyset$ has too low of a timestamp). In Figure 2, the versions returned by T_2's second round of reads ($\{x_1, y_1\}$) are a companion set. If a third transaction T_3 wrote another version x_3, x_3 would be a companion version to y_1 (because x_3 has higher version than x_1). 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). *In the absence of G1c phenomena, if the set of versions read by a transaction is a companion set, the transaction does not exhibit fractured reads.*

Claim 1 follows from the definitions of companion sets and fractured reads.

JUSTIFICATION. If V is a companion set, then every version $x_i \in V$ is a companion to every other version $y_j \in V$ that 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 in the following:

CLAIM 2. RAMP-F *provides RA isolation.*

JUSTIFICATION. Each version in RAMP-F contains information regarding its sibling items, which can be identified by item and timestamp. Given a set of 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 nonfinal (intermediate) writes. Moreover, RAMP-F timestamps are assigned on a per-transaction basis, preventing write-dependency cycles and therefore *G1c*. This establishes that RAMP-F provides RA. □

RAMP-F Scalability and Independence. RAMP-F also provides the independence guarantees from Section 3.5. 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 *is coordination-free.*

Recall from Section 3.5 that coordination-free execution 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).

JUSTIFICATION. Clients in RAMP-F do not communicate or coordinate with one another and only contact servers. Accordingly, to show that RAMP-F provides coordination-free execution, 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 does not depend on the success or failure of other clients' RAMP-F operations. □

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

JUSTIFICATION. 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 RA isolation.*

JUSTIFICATION. To show that RAMP-S provides RA, it suffices to show that RAMP-S second-round reads (*resp*) are a companion set. Recall that all versions produced in a RAMP-S transaction are assigned the same timestamp. 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 ensures coordination-free execution and partition independence. The proofs closely resemble those of RAMP-F: Invariant 1 ensures that incomplete commits do not stall readers, and all server-side operations are guaranteed to complete.

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 RA isolation.*

JUSTIFICATION. To show that RAMP-H provides RA 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 nonsibling 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 coordination-free execution 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 previously, these empty responses correspond to false positives in the Bloom filter and therefore do not affect correctness.

ACKNOWLEDGMENTS

The authors would like to thank Peter Alvaro, Rick Branson, Neil Conway, Aaron Davidson, Jonathan Ellis, Mike Franklin, Tupshin Harper, T. Jake Luciani, Aurojit Panda, Nuno Preguiça, Edward Ribeiro, Mehul Shah, Shivaram Venkataraman, and the SIGMOD and TODS reviewers for their insightful feedback.

REFERENCES

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

Atul Adya. 1999. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. Ph.D. Dissertation. MIT.

Divyant Agrawal and Vasudha Krishnaswamy. 1991. Using multiversion data for non-interfering execution of write-only transactions. In *SIGMOD*.

Hagit Attiya and Jennifer Welch. 2004. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. (2nd. ed.). John Wiley Interscience.

Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2014a. Highly available transactions: Virtues and limitations. In *VLDB*.

Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2014b. Highly available transactions: Virtues and limitations (extended version). arXiv:1302.0309.

Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2015. Coordination avoidance in database systems. In *VLDB*.

Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2012. The potential dangers of causal consistency and an explicit solution. In *SOCC*.

Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2014c. Scalable atomic visibility with RAMP transactions. In *SIGMOD*.

Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013. Bolt-on causal consistency. In *SIGMOD*.

Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. 2014d. Quantifying eventual consistency with PBS. *The VLDB Journal* 23, 2 (2014d), 279–302.

Jason Baker, Chris Bond, James Corbett, J. J. Furman, and others. 2011. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*.

H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. 1995. A critique of ANSI SQL isolation levels. In *SIGMOD*.

Phillip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, New York.

Ken Birman, Gregory Chockler, and Robbert van Renesse. 2009. Toward a cloud computing research agenda. *SIGACT News* 40, 2 (June 2009), 68–80.

Jose A. Blakeley, Per-Ake Larson, and Frank Wm. Tompa. 1986. Efficiently updating materialized views. In *SIGMOD*.

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

Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chukka, Peter Dimov, and others. 2013. TAO: Facebook's distributed data store for the social graph. In *USENIX ATC*.

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

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, and others. 2006. Bigtable: A distributed storage system for structured data. In *OSDI*.

Bernadette Charron-Bost. 1991. Concerning the size of logical clocks in distributed systems. *Information Processing Letters* 39, 1 (July 1991), 11–16. DOI:http://dx.doi.org/10.1016/0020-0190(91)90055-M

Rada Chirkova and Jun Yang. 2012. Materialized views. *Foundations and Trends in Databases* 4, 4 (2012), 295–405.

Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, and others. 2008. PNUTS: Yahoo!'s hosted data serving platform. In *VLDB*.

Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *ACM SOCC*.

James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, and others. 2012. Spanner: Google's globally-distributed database. In *OSDI*.

Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. 2010. Schism: A workload-driven approach to database replication and partitioning. In *VLDB*.

Sudipto Das, Divyant Agrawal, and Amr El Abbadi. 2010. G-store: A scalable data store for transactional multi key access in the cloud. In *ACM SOCC*.

K. Daudjee and K. Salem. 2004. Lazy database replication with ordering guarantees. In *ICDE*. 424–435.

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

Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Communications in ACM* 56, 2 (2013), 74–80.

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, and others. 2007. Dynamo: Amazon's highly available key-value store. In *SOSP*.

Jose Faleiro, Alexander Thomson, and Daniel J. Abadi. 2014. Lazy evaluation of transactions in database systems. In *SIGMOD*.

Alan Fekete, David Gupta, Victor Luchangco, Nancy Lynch, and Alex Shvartsman. 1996. Eventually-serializable data services. In *PODC*.

Seth Gilbert and Nancy Lynch. 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News* 33, 2 (2002), 51–59. DOI:http://dx.doi.org/10.1145/564585.564601

Jim Gray and Leslie Lamport. 2006. Consensus on transaction commit. *ACM TODS* 31, 1 (March 2006), 133–160.

J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. 1976. *Granularity of Locks and Degrees of Consistency in a Shared Data Base*. Technical Report. IBM.

Pat Helland. 2007. Life beyond distributed transactions: An apostate's opinion. In *CIDR*.

Sean Hull. 2013. 20 obstacles to scalability. *Communications of the ACM* 56, 9 (2013), 54–59.

Nam Huyn. 1998. Maintaining global integrity constraints in distributed databases. *Constraints* 2, 3/4 (Jan. 1998), 377–399.

Evan P. C. Jones, Daniel J. Abadi, and Samuel Madden. 2010. Low overhead concurrency control for partitioned main memory databases. In *SIGMOD*.

Robert Kallman, H. Kimura, J. Natkins, A. Pavlo, and others. 2008. H-store: A high-performance, distributed main memory transaction processing system. In *VLDB*.

Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. 1992. Providing high availability using lazy replication. *ACM Transactions on Computer Systems* 10, 4 (Nov. 1992), 360–391. DOI:http://dx.doi.org/10.1145/138873.138877

Avinash Lakshman and Prashant Malik. 2008. Cassandra+—A decentralized structured storage system. In *LADIS*.

Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM* 21, 7 (July 1978), 558–565. DOI:http://dx.doi.org/10.1145/359545.359563

Richard J. Lipton and Jonathan S. Sandberg. 1988. *PRAM: A Scalable Shared Memory*. Technical Report TR-180-88. Princeton University.

Francois Llirbat, Eric Simon, Dimitri Tombroff, and others. 1997. Using versions in update transactions: Application to integrity checking. In *VLDB*.

Wyatt Lloyd, Michael J. Freedman, and others. 2011. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *SOSP*.

Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2013. Stronger semantics for low-latency geo-replicated storage. In *NSDI*.

Dahlia Malkhi, Michael K. Reiter, Avishai Wool, and Rebecca N. Wright. 2001. Probabilistic quorum systems. *Information and Computation* 170, 2 (2001), 184–206.

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

Moni Naor and Avishai Wool. 1998. The load, capacity, and availability of quorum systems. *SIAM Journal on Computing* 27, 2 (1998), 423–447.

Andrew Pavlo, Carlo Curino, and Stanley Zdonik. 2012. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *SIGMOD*.

Daniel Peng and Frank Dabek. 2010. Large-scale incremental processing using distributed transactions and notifications. In *OSDI*.

Shirish Hemant Phatak and B. R. Badrinath. 1999. Multiversion reconciliation for mobile databases. In *ICDE*.

Lin Qiao, Kapil Surlaker, Shirshanka Das, Tom Quiggle, and others. 2013. On brewing fresh Espresso: LinkedIn's distributed data serving platform. In *SIGMOD*.

Nicolas Schiper, Pierre Sutra, and Fernando Pedone. 2010. P-store: Genuine partial replication in wide area networks. In *IEEE SRDS*.

Marc Shapiro and others. 2011. *A Comprehensive Study of Convergent and Commutative Replicated Data Types*. Technical Report 7506. INRIA.

Jeff Shute and others. 2013. F1: A distributed SQL database that scales. In *VLDB*.

Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. 2011. Transactional storage for geo-replicated systems. In *SOSP*.

Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. 2001. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*.

Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theimer, and Brent B. Welch. 1994. Session guarantees for weakly consistent replicated data. In *PDIS*.

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

Kevin Weil. 2011. Rainbird: Real-time analytics at Twitter. (2011). Strata Conference, http://slidesha.re/hjMOui.

Stanley B. Zdonik. 1987. Object-oriented type evolution. In *DBPL*. 277–288.

Jingren Zhou and others. 2007. Lazy maintenance of materialized views. In *VLDB*.

Received February 2015; revised February 2016; accepted March 2016