Original Paper Number 60

To the SIGMOD 2014 reviewers and editor: Thank you for your feedback. As we discuss below, we have integrated your comments and believe that they have materially strengthened the submission. We briefly discuss structural changes related to your feedback before addressing individual comments in the remaining pages of this front matter:

- Given feedback from Reviewers Two and Three regarding our intended use cases and motivation, we have shortened the motivation section (and now begin with the System Model on Page 3). We have also added a new section (Section 5) that is specifically devoted to explaining how the RAMP API and implementation provides sufficient concurrency control for these use cases. We hope that this new section provides additional clarity by demonstrating *how* RAMP concurrency control addresses our motivating use cases.

- Given feedback from Reviewers One and Three regarding replication and fault tolerance, we have expanded the original discussion in Section 4 to an entire subsection on the topic (Section 4.5). We discuss replication strategies and discuss (and evaluate in Section 6) the use of the Cooperative Termination Protocol algorithm (described in [Bernstein et al. 1987]) to complete stalled/"blocked" writes.

- Given feedback from Reviewers One and Two regarding subtleties in the proposed RA semantics (e.g., transitivity), we have added a discussion section (Section 3.2) to specifically address these issues and discuss when RA is both sufficient and insufficient for applications.

- As noted below, we have clarified the text in several places to address concerns raised by the reviewers and, in general, to improve readability.

Overall, we believe that these changes (and those described below) satisfy the Meta-Review request that we "address the concerns identified in Reviewer 1's and Reviewer 3's responses to question Q7, as well as the concerns raised by Review 2 concerning the evaluation and related work."

In-line replies to specific comments follow. Thank you again for your consideration of our manuscript.

The Authors

Reviewer 1 Comments

> Include further discussion of replication and durability.

Response: As noted above, we have added a new subsection (Section 4.5) to specifically address replication strategies and durability.

> Explain how all shards eventually learn the commit decision in the case where the commit loop is non-atomic (e.g., due to a client failure in the middle of it). Unless there is something like 2PC or Paxos commit happening here, there is a clear durability hazard.

Response: In the new subsection on replication and durability we have also added a discussion of the (existing) Cooperative Termination Protocol algorithm and evaluate its performance in Section 6.

> Extend/clarify discussion of how read-only and write-only transactions are composed into RMW transactions. A short example would be helpful here, and could help illustrate both the strengths of RA semantics and its limitations.

Response: We have made two changes. First, we have discussed the programmability of read/write transactions in a larger section on limitations of RAMP transactions (and give several examples; Section 3.2). Second, we have revised the original discussion of how to actually build read/write transactions using the initial formulation of RAMP transactions (Section 4.4).

> Eschewing transitivity seems key to synchronization/partition independence, but is also a BIG limitation. Explicit discussion of this issue is needed. Giving example applications that do/don't rely on transitivity would be helpful here. (E.g.: In an antisocial network, if "enemy" graph edges are inserted and read with RA isolation across users, then enemy relationships will always appear symmetrical but enemy-of-enemy (a.k.a. "friend") relationships may not.)

Response: This is an excellent point that we have now made explicit in Section 3.2. We use a simple example of comment threads in a social network to show when happens-before is required for correct behavior. In general, we do not believe (but leave it to conjecture that) ensuring happens-before across arbitrary items is achievable with partition independence.

> Insufficient discussion of durability/causality. The two-phase write protocol prevents data loss, but it appears that a commit decision can be forgotten after being observed by a reader. (Example: T1 writes A1 and B1 (at different partitions). T1 completes the commit phase at A's partition. T2 reads A and B, sees A1 and B1, then reads only B and sees $\perp$)

Response: This is an excellent point, and one that was lost in the prior discussion of PRAM semantics. We have clarified this discussion (now in Section 3.2) to address transitivity. The above anomaly cannot occur under RAMP algorithms when "Faster commit detection" is enabled, but another, which we discuss in the revised section, could.

> More/clearer discussion of replication is needed.

Response: As above, we have added Section 4.5 to specifically address distribution and replication.

```
The lack of transitivity seems like a major drawback.  It may indeed be
that the target applications tolerate this just fine, but more discussion is
definitely needed.
```

Response: We believe that Section 3.2 and the additional discussion in Sections 2 and 5 elucidate this point. RA is not a panacea and we have made a concerted effort to reiterate this in our revisions.

```
The plots are really hard to read.  Please make them taller...
```

Response: We have increased the figure size and have also modified the line width, font size, and marker size for legibility.

```
...and use actual log scales where appropriate ("1, 100, 1000" is NEVER okay).
```

Response: We have added the additional points to Figure 3 and apologize for the original (unintended) omission.

```
Figure 4 also needs an accompanying plot with per-node throughput in order to
be easily readable.
```

Response: We have added the suggested accompanying plot to Figure 4.

Thank you for your comments.

Reviewer 2 Comments

```
* The initial part of the paper is too verbose -- 3-4 pages (33% of the paper)
is motivational material to position the proposed RAMP model.
```

Response: Per your suggestion, we have revised the first several pages with an eye towards brevity and now end the motivation on page 2.

```
* The model itself is stated in a few lines.  This model is closely related to
"snapshot isolation" -- but the authors do not make any attempt to highlight
the similarities and differences between the two.
```

Response: We have provided additional discussion of the model in relation to other proposed models in Section 3.1 and believe the new discussion in Section 3.2 also highlights when RA is sufficient and when it is not (i.e., when Snapshot Isolation or a related, stronger model is necessary for correctness).

```
* the illustration in Figure 1 is confusing -- since T2 on Py reads uncommitted
data (granted data is prepared -- but its fate has not yet been determined).
```

Response: One of the key contributions to RAMP scalability is clients' ability to determine when a Read Atomic set of versions has been assembled; Figure 1 depicts this process. In the example, Py has not yet received a COMMIT for y_1 but, because RAMP-L client C2 has requested ts_{req}=1, T1 must have committed on some participant (in the example, Px). We have clarified the introductory text in Section 4 and Section 4.1 in order to further illustrate this point.

```
* Restriction of update transactions being write-only is highly contrived -- if
anything this is a fatal flaw in the paper.
```

Response: While RA isolation supports read-write transactions, it indeed does not provide the same guarantees for updates as in stronger models like Snapshot Isolation or Serializability (namely, concurrent writers can proceed concurrently). This is key to achieving synchronization independence and is an important component of RAMP scalability. Of course, as we note in Section 1, RA guarantees are not suitable for all applications, but they match those that that we have identified in Section 2. The addition of Section 3.2 should assist readers in identifying when these semantics are sufficient, and the discussion in Section 5 helps clarify how they are useful in our motivating examples.

```
Given this restriction, it is unfair to use standard concurrency control
protocols for your evaluation.  In fact, there are publications that have
considered the issue of synchronizing write-only transactions (but these
protocols do not rule out other types of transactions).
```

Response: The reviewer brings up an excellent point that a comparison to, say, serializable transactions is not sufficient to demonstrate the performance benefits of RAMP. We have compared to three lock-based policies (one of which, `LWLR` provides *PL-2.99* but two of which, `LWSR` and `LWNR`, are weaker than RA isolation and highlight overheads due to read lock contention). However, we have also compared to a baseline of no isolation (`NWNR`, the primary focus of our discussion) and the most scalable write-only transaction mechanism we have found in the literature, Eiger (`MSTR`), which was proposed in NSDI 2013 and, as we discuss, generalizes Chan and Gray's classic Read-only Transactions and G-Store's dynamic entity grouping to write-only transactions. We believe that these five algorithms—only two of which provide at least RA semantics and only three of which use

locks—are a reasonable basis for comparison to our three proposals (RAMP-L, RAMP-S, RAMP-B).

As the reviewer mentions, many prior publications have considered write-only and read-only transactions. We have added the reviewer's suggested references to the related work section. We believe that the MSTR/Eiger protocol exhibits the best scalability for existing write-only transactions that provide at least RA isolation due to synchronization independence and lack of centralized coordinator. Of course, neither MSTR or RAMP are serializable, so the suggested references (namely [Agrawal and Krishnaswamy 1991]) provide a useful point of comparison should readers desire stronger semantics (as discussed in Section 3.2).

> The entire evaluation is a suspect. Given that your reads can have high overhead -- it is surprising that you are within 8% of NWNR. The only explanation is that in your baseline experiment -- the contention is so low that effectively there are no collisions -- and effectively your RAMP-L protocol becomes almost like NWNR (don't need experimental evaluation for this you can analytically establish that).

Response: The reviewer provides an excellent point that RAMP-L and RAMP-B transactions are carefully constructed to incur no overhead when there are no concurrent writes to a given item. The number of concurrent writes dictates the number of RAMP-L and RAMP-B reads that will require two RTTs; as database size decreases and/or the number of writes increases, RAMP-L and RAMP-S throughput will decrease. We have illustrated this phenomenon analytically in Table 1 and experimentally in our discussion of "Read Proportion" and "Database Size" in Section 6 and Figure 3 (e.g., the first plot in Figure 3 and accompanying discussion demonstrate that, with 100% writes, the throughput of all RAMP algorithms is 65% of NWNR). As Table 1 and our experimental results demonstrate, RAMP-B is an effective lower bound on RAMP-L and RAMP-S throughput as it *always* incurs two RTTs for both reads and writes; we make no attempt to conceal this fact and we have attempted to make this fact more explicit in our revisions.

We have chosen a default read/write ratio of 95 to 5. We have added additional discussion of this decision in Section 6.1: many use cases identified by industry (e.g., web and social networking) are read-mostly. For example, the authors of Espresso note a target workload of $1000:1$—fifty times fewer writes than we target in our baseline workload—and Spanner reports $3396:1$.

> Read Proportion analysis. Don't understand "as the proportion of reads decreases, the cost of RAMP transaction decreases" -- but why is the throughput declining if transactions cost less? Second, as the proportion .... (supports my observation above regarding your baseline set-up -- essentially you may have RAMP-X -- but that protocol is behaving as NWNR).

Response: The reviewer makes a sound observation. As Table 1 states analytically and Figure 3 verifies, this sentence should have read that "the cost of RAMP transactions **increases**" (emphasis added). We decided to rewrite this section to avoid any ambiguity. The new text should be clearer and reflects the reviewer's observation above. We apologize for the confusion. We view RAMP-S as a lower bound on throughput, while Figure 3 demonstrates several clear trade-offs in this space: the write proportion determines how often reads require two RTTs. We have also revised our final discussion at the end of Section 6.2 in order to better reflect these concerns.

```
The experiment on "Database Size" is at odds with the analysis in "Read
Proportion".  The fact that throughput curves remain almost flat across all
database sizes -- is a bit alarming.  Most likely your write activity is so
miniscule that it only effectively you are processing reads and hence data size
does not matter.
```

Response: As noted, we have modified the earlier analysis of Read Proportion. Figure 3 ("Database Size") demonstrates throughput decreases as database size shrinks: compared to NWNR, RAMP-L throughput decreases by 10% and RAMP-B throughput decreases by 16%. As noted above, Figure 3 also discusses performance under write-heavy workloads. To reiterate the above points, we have added additional emphasis in the text that RAMP-S is a lower bound on throughput.

Double-blind reviewing unfortunately prohibits us from sharing our reference implementation that we plan to release under a permissive open source license following the reviewing process. In addition to assisting with reproducibility, this will allow readers to evaluate our algorithms under the YCSB framework and to perform additional experimentation if desired.

```
Missing references.
```

```
There is some prior work on write-only transactions as well as maintenance of
derived data -- these references should be cited.
```

```
Franois Llirbat, Eric Simon, Dimitri Tombroff:  Using Versions in Update
Transactions:  Application to Integrity Checking.  VLDB 1997:  96-105
```

```
Nam Huyn:  Multiple-View Self-Maintenance in Data Warehousing Environments.
VLDB 1997:  26-35
```

```
Nam Huyn:  Maintaining Global Integrity Constraints in Distributed Databases.
Constraints 2(3/4):  377-399 (1997)
```

```
Shirish Hemant Phatak, B. R. Badrinath:  Multiversion Reconciliation for Mobile
Databases.  ICDE 1999:  582-589
```

```
D. Agrawal, V Krishnaswamy.  Using multiversion data for non-interfering
execution of write-only transactions.  Proceeding SIGMOD '91 Proceedings of
the 1991 ACM SIGMOD international conference on Management of data.
```

Response: We have noted these references, citing them each of them, in Section 7 and in the remainder of the text as appropriate. As a general note, and, as we discuss in the revision, we are not aware of an existing synchronization and partition independent mechanism that provides RA isolation or stronger.

Thank you for your comments.

Reviewer 3 Comments

> (1) The paper needs to discuss the anomalies above and propose some solutions.

Response: We discuss each of the anomalies in turn.

> There is a bug in the protocol: It implements neither per-partition monotonicity nor eventual consistency! Assume a writer writes tuples a and b to partitions A and B, respectively, and it crashes in the loop in Algorithm 1, Line 20, so that A gets the commit message, but B does not.

> Anomaly 1: If no transaction ever reads both a and b, then A will reflect the updates for all time, but B will never reflect the writer's updates. This does not violate read atomicity, but it does violate eventual Consistency (with a big "C", like in ACID).

Response: The reviewer poses an excellent question: what happens during stalled commits? We have added a discussion (and, later, evaluation) of the Cooperative Termination Protocol from Bernstein et al.'s book to address this concern. In the anomaly above, B will time out, contact A, learn that A has committed the transaction, and will commit its pending writes. The CTP protocol allows partitions to complete any partial commits, providing the suggested eventual Consistency.

> Anomaly 2: At some point, A and B will garbage collect old versions of their state. Since b never committed, the unwritten value will be deleted. Later, when a reader attempts to read both a and b, it will observe the old and new values at the same time. It is not sufficient to simply commit prepared messages during GC, since some may be from doomed transactions.

Response: The reviewer makes a keen observation that uninhibited Garbage Collection may violate RA isolation. In our discussion of Garbage Collection, we note that a version can be discarded as long as no readers will attempt to read it in the future. This precludes discarding uncommitted data—a subtlety that we have made more explicit in our revision of this discussion.

This raises another question: how long will servers have to retain stalled writes? As we discuss in the revision, the CTP protocol above addresses this concern in the case of partial prepares and partial commits. In the event that all partitions have prepared but none has committed, just as in traditional 2PC, the partitions will have to retain the data.

> Anomaly 3: Assume the optimization "Faster Commit Detection" is not implemented (the optimization will fix this anomaly, but not Anomalies 1, 2 or 4). One class of reader always reads both A and B, but another class only reads B. The two classes of readers will never agree on the current value of B. This is a violation of both monotonicity and eventual consistency (with a little "c", like in NoSQL).

Response: In addition to the "Faster Commit Detection" mechanism, the CTP protocol described above (and discussed in the revision) will also address this anomaly: B will contact A to learn about the commit decision and will eventually serve the correct version of B.

> There is an additional problem that doesn't rely upon a client that crashes mid-write:

> Anomaly 4: One of the use cases is "derived data maintenance", but I don't see how this can work with lost writes anomalies. In particular, even when the dervied data is a counter (so, you get commutativity), algorithm 1, line 8 can drop increments performed by slow-running transactions (since a faster transaction could perform the increment, causing the "updateIfGreater" to become a no-op). This could be fixed by special-casing commutative operations, but it is unclear that it will work correctly in the face of more general derived data maintenance operations.

Response: We have addressed this concern in the newly added Section 6. Previously, we discussed derived data in several places and, upon reviewing these comments, decided to cleanly separate the discussion into two places: motivating examples (Section 2) and clear application of the RAMP algorithms (Section 5). This should hopefully clarify our discussion of derived data. RAMP addresses the problem of maintaining derived data by providing concurrency control that ensures base data can be observed only with any modifications to derived data. We have specifically addressed the counter anomaly in Section 5, and the CRDT G-Counter (as adopted in Riak) we discuss is the best solution to handling concurrent increment and decrement operations that we are aware of.

As we reiterate in the text, we view our primary contribution as providing a new solution to the problem of concurrency control over views rather than to the large body of work pertaining to actual computation of materialized views (to which we have added several references). We hope that the revised text clarifies this point.

> It seems like (1) and (2) can be fixed by adding a hook to the GC logic to have it detect prepared writes, and to attempt to finish the transaction.

> However, a GC-based fix might be tricky. Assume that a has been overwritten a second time, and A GCed its old versions of a. Later, when B decides to GC the prepared version of b, A might not remember whether or not the transaction that wrote to both a and b committed. A could have told B that the transaction committed when the old a was GCed, but from A's perspective, the write to a is just a normal committed write, so if it was going to send the commit out to B, it would need to do it for every committed item it collects.

> Alternatively, you could add a third (asynchronous) round to each transaction, at which point the protocol becomes extremely similar to 3PC: The main difference (other than the motivation) is that, in the "finalizing commit" phase, timeouts cause commits instead of aborts.

Response: The CTP protocol coupled with garbage collection of committed data handles these cases without requiring broadcast on commit or requiring clients to perform 3PC-like third-rounds.

> (2) The experiments need to pick one of the solutions, and include numbers that reflect the overhead of ensuring eventual consistency (and, optionally, derived data maintenance). If fixing the anomalies significantly impacts performance, then it would make sense to leave the current numbers in as a comparison point with the new numbers (so readers can decide how much they really care about eventual consistency, etc...).

Response: We have included an evaluation of CTP in the Section 6 under a varying proportion of incomplete/simulated "blocked" writes. As we discuss, CTP only runs when writes are blocked or time-outs fire, making its average-case performance impact negligible. Nevertheless, we have

included numbers for very high failure rates to allow the reader to decide whether to apply the linear performance penalty associated with CTP (relevant if, say, tens of percent of transactions fail mid-commit—a scenario we believe is unlikely and indicative of larger systemic problems).

> These fixes are involved enough to make me nervous about their correctness, and about the overhead they would have in the experiments.

Response: We believe that the CTP protocol and our experimental evaluation address this concern given expected abort rates.

> Also, the paper's treatment of fault tolerance is pretty sloppy. Consider the optimization "Metadata garbage collection". Assume that, as that section suggests, the client returns before sending commits out to all servers, and later, one server crashes. Now what? Can the other servers assumed the write succeeded? If not, then the client incorrectly returned success. If so, and the crashed server never got its prepare, then atomicity will be violated when the crashed server recovers.

Response: The reviewer brings up a valid concern that committing on some servers without preparing on others could lead to anomalies.

In the specific case of "Metadata garbage collection", we observe that, "once all of a sibling's versions are reflected in their respective partitions' $lastCommit$, there is no requirement for a second round of communication in our algorithms." This precondition is only met after all servers have performed COMMIT and therefore the observation of reduced RTTs only applies to GET_ALL operations—writes will still require a second round of communication. We apologize for the ambiguity and have made this point more explicit in the revision.

Regarding the specific concern that some servers might have performed COMMIT while others (i.e., the crashed server above) never performed PREPARE, we note that, due to the structure of the two-phase PUT_ALL protocol, COMMIT will only invoked after all servers have performed PREPARE. The scenario described is not possible in the RAMP algorithms. The protocol may stall in the case of a failed server but does not violate correctness (i.e., safety).

As noted above, we hope that the new Section 4.5 will address any concerns regarding durability and fault tolerance both of clients and partitions.

> Along these lines, it might help the presentation if there was a discussion of the window of vulnerability in 2PC, and an explanation of how 3PC and RA avoid it.

Response: Along with our discussion of CTP, we have discussed when exactly writes may become "blocked" (when a violation of the pre-conditions of partition independence are violated—at least one partition is unreachable—and when a client dies between PREPARE and COMMIT). We do not rely on synchronous networks in order to maintain write availability as in 3PC.

> Finally, I don't really understand how you compute the happens-before relation over the timestamps, or how companion sets are passed around / computed during reads. One possibility would be to include a list of old timestamps for each returned item in the first round, but it's unclear if that's what's going on.

Response: The computation of the companion sets depends on the algorithm (e.g., RAMP-L's v_{latest} computation computes the companion sets). We have modified the text to highlight exactly where this computation occurs in each algorithm.

Additionally, we have revised the introduction to Section 4, which we believe provides a friendlier introduction to the RAMP algorithms.

```
(3) An explanation of how fault tolerance / replication interacts with the
protocol is also needed.
```

Response: Our introduction of Section 4.5 is intended to address this concern.

Thank you for your comments.

Read Atomic Multi-Partition Transactions

Original Paper Number 60

ABSTRACT

Distributed databases can provide scalability by partitioning data across several servers. However, multi-partition, multi-operation transactional access is often expensive, employing coordination-intensive locking, scheduling, or validation mechanisms. Accordingly, many real-world systems completely eschew semantics for multi-partition operations. This leads to inconsistency for a large class of applications that require atomically isolated cross-partition update and access, including secondary indexing, foreign key enforcement, and derived data maintenance. In this work, we identify a new isolation model—Read Atomic (RA) isolation—that matches the requirements of our target use cases: either all or none of each transaction's updates are observed. This allows correct behavior for a large class of currently underserved applications without compromising scalability or availability. We subsequently present algorithms for Read Atomic Multi-Partition (RAMP) transactions that provide RA isolation while offering guaranteed commit despite partial failures (via synchronization independence) and minimized communication between servers (via partition independence). We demonstrate that, in contrast with existing algorithms, RAMP transactions incur limited overhead—even under high contention—while scaling linearly to 100 servers.

1. INTRODUCTION

Faced with growing amounts of data and unprecedented query volume, distributed databases increasingly spread their data across multiple servers, or *partitions*, such that no one partition contains an entire copy of the database [7, 13, 18, 19, 22, 30, 43]. This strategy succeeds in allowing near-unlimited scalability for operations that access single partitions. However, database partitioning substantially complicates operations on items located on multiple partitions: unlike a single-site database, a partitioned database supporting multi-partition operations must coordinate between multiple servers—often synchronously—in order to provide queries with correct responses [17, 29, 30, 36].

In this work, we address a largely underserved class of applications requiring multi-partition, atomically (all-or-nothing) isolated[1] *transactional* access. The status quo for multi-partition atomic transactions results in 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. Modern real-world systems commonly opt for protocols that guarantee fast and scalable operation but provide few—if any—transactional semantics for operations on arbitrary sets of data items [11,13,15,22,26,39,44]. This results in application-level inconsistency for use cases that require atomic transactional access, which include secondary indexing, foreign key constraint enforcement, and derived data maintenance (Section 2). In contrast, many traditional transactional mechanisms ensure atomicity of updates [8, 17, 43]. However, these algorithms—such as two-phase locking and variants of optimistic concurrency control—are often coordination-intensive, slow, and, under failure, unavailable in a distributed environment [1, 5, 18, 29, 36]. This dichotomy between scalability and atomic transactional support has been described as "a fact of life in the big cruel world of huge systems" [25]. The proliferation of non-transactional multi-item operations is endemic to a widespread "fear of synchronization" at scale [9].

Our contribution in this paper is to demonstrate that atomically isolated transactions on partitioned databases are *not* at odds with availability or performance. To do so, we first propose a new, non-serializable isolation model—Read Atomic (RA) isolation—which provides acceptable semantics for our target use cases. Specifically, RA isolation ensures that all or none of each transactions' updates are visible to others and that each transaction reads from an atomic snapshot of database state (Section 3). While several existing but stronger isolation models, such as Repeatable Read (*PL-2.99* [2]), are sufficient for our target applications, they suffer from poor performance and unavailability in the presence of network partitions and partial failure [1]. In contrast, weaker isolation models such as Read Committed (*PL-2* [2]) isolation are scalable and available under failures [5] but are too weak and allow atomicity anomalies. RA provides the correct semantics for our target use cases, with little overhead over weaker semantics (e.g. *PL-2*) and substantially less overhead compared to stronger semantics (e.g. *PL-2.99* or serializability). This custom-tailoring inherently means that RA isolation is *not* sufficient for all application scenarios (e.g., use cases sensitive to concurrent updates like Write Skew [1,2]), but, for the many that it fits, RA concurrency control provides scalability advantages.

We present three new, scalable algorithms for achieving RA isolation that we collectively title Read Atomic Multi-Partition (RAMP) transactions (Section 4). RAMP transactions guarantee scalability and outperform existing algorithms because they satisfy two key scalability constraints. First, RAMP transactions guarantee *synchronization independence*: one client's transactions cannot cause another client's transactions to stall or fail. Second, RAMP transac-

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

XXX YYY

tions guarantee *partition independence*: clients never contact partitions that their transactions do not directly access. Together, these properties ensure guaranteed termination, limited coordination across partitions, and horizontal scalability for multi-partition access. Our algorithms provide a trade-off between metadata and performance. RAMP-S requires constant space and no more than two round trip time (RTT) delays for reads, while RAMP-L requires linear space and expected one RTT for reads and RAMP-B employs Bloom filters [10] to provide an intermediate solution. In addition to providing a theoretical analysis and correctness proofs, we demonstrate that RAMP transactions incur limited overhead compared to inconsistent system configurations without any concurrency control (Section 6). We also demonstrate linear scalability to over 7 million operations per second on a 100 node cluster (at a $< 5\%$ cost to peak throughput) as well as substantial performance gains compared to several existing approaches.

2. MOTIVATING EXAMPLES

In this paper, we will study atomically visible modifications to multiple records—use cases where either all updates should be visible, or none should be. In this section, we outline several specific cases that will drive our choice of semantics in Section 3 and which we will revisit in Section 5.

Multi-entity updates. To begin, as a simple example, consider a social networking application: if two users, Sam and Mary, become "friends" (a bi-directional relationship), an observer should not see that Sam is a friend of Mary but Mary is not a friend of Sam. In general, many applications such as those using LinkedIn's Espresso "need guarantees that any updates that apply to [multiple] Entities must eventually happen" and that a "few cases...require both the edges to be modified atomically" (i.e., not "eventually"). Percolator's authors at Google observe that "transactions make it more tractable for the user to reason about the state of the system and to avoid the introduction of errors into a long-lived repository," specifically citing an example of making sure that new documents and their hashes are made atomically visible [37].

Based on practitioner accounts, we further classify three classes of use cases that benefit from atomically visible multi-item updates:

1.) Foreign key constraints. Many database schemas contain information about relationships between records in the form of foreign key constraints. For example, Facebook's TAO [11], LinkedIn's Espresso [39], and Yahoo! PNUTS [15] store information about business entities such as users, photos, and status updates as well as relationships between them (e.g., the friend relationships above). Their data models often represent bi-directional edges as two distinct uni-directional relationships. For example, in TAO, "although associations are directed, it is common for an association to be tightly coupled with an inverse edge:" a user "liking" a given page updates both the `LIKES` and `LIKED_BY` associations [11]. PNUTS's authors describe an identical scenario [15]. These scenarios require foreign key maintenance and often, due to their unidirectional relationships, general-purpose multi-entity update and access. Atomicity violations surface as broken bi-directional relationships (e.g., friendship is bi-directional, but, while Sam is a `friend_of` Mary, it appears Mary is not a `friend_of` Sam) and dangling or incorrect references (e.g., Frank is an `employee` of `department.id=5`, but no such `department` exists in the `department` table).

2.) Secondary indexing. Data is typically allocated between partitions according to a primary key (e.g., user ID). This allows fast location and retrieval of data via primary key lookups but makes access by secondary attributes (e.g., birth date) challenging. There are two dominant strategies for distributed secondary indexing. First, the *local secondary index* approach partitions secondary index data structures by primary key, co-locating each secondary index partition with its corresponding primary data [7, 39]. This allows easy, single-server update but requires contacting every partition for lookups by secondary attributes (write-one, read-all). Alternatively, the *global secondary index* approach locates secondary indexes (which are still possibly partitioned, but by a secondary attribute) separately from primary data [7, 15]. This allows fast secondary lookups (read-one) but multi-partition update (write-two).

For read-dominated workloads [11], global secondary indexing is preferable, but, given the expense of existing protocols, real-world services employ either local secondary indexing (e.g., LinkedIn Espresso [39] and Google Megastore's [7] local indexes, Riak, and Cassandra) or non-atomic global secondary indexing (e.g., Espresso and Megastore's global indexes, Yahoo! PNUTS's secondary index proposal [15]). Local secondary indexing requires that all partitions respond to all secondary lookup, so adding more partitions will not improve read throughput. In contrast, non-atomic global secondary indexing allows fast lookups but may return incorrect data. For example, in a database partitioned by `id` with a non-atomically maintained secondary index on `salary`, the result set for the query `'SELECT id, salary WHERE salary > 60,000'` might contain entries with `salary` less than \$60,000 and omit some items with `salary` greater than \$60,000.

3.) Derived data maintenance. It is common to precompute queries over data, as in Twitter's Rainbird service [44], Google's Percolator [37], and LinkedIn's Espresso systems [39]. As a simple example, Espresso stores a mailbox of messages for each user along with statistics about the mailbox messages: for a read-mostly workload like the mailbox application, it is more efficient to maintain (i.e., pre-materialize) a count of unread messages rather than scan all messages every time a user accesses her mailbox [39]. In this case, any unread message indicators should remain in sync with the messages in the mailbox. However, atomicity violations will allow derived data to diverge from the base data (e.g., Susan's mailbox displays a notification that she has `unread` messages but all 63,201 messages in her `inbox` are marked as `read`). Regulating the visibility of changes to primary data and (regardless of computation strategy) derived data is frequently important for user experience.

Status quo. As we discuss in Section 7, few large-scale production systems provide multi-item atomic transactions. For example, Facebook's TAO "does not provide atomicity between the two updates. If a failure occurs [one association] may exist without an inverse; these hanging associations are scheduled for repair by an asynchronous job" [11]. The Espresso authors plan to add support for global secondary indexing and "cross entity transaction support in the future" but do not currently provide them and, due to lack of support, Espresso users currently "risk seeing inconsistent data at times" [39]. In response to customer requests, PNUTS added support for "bundled updates, provid[ing] atomic, non- isolated updates to multiple records...[that] are guaranteed to eventually complete, but other transactions may see intermediate states resulting from a subset of the updates" [15]. The PNUTS authors state that "further research is needed to examine the semantic implications of answering queries using [our] possibly stale indexes and views" [15]. Data stores like Bigtable [13], Dynamo [22] and popular open source "NoSQL" stores [35] like Riak and Cassandra completely eschew guarantees on multi-key operations.

These systems all provide excellent scalability at the expense of multi-partition transactional semantics. Our goal in the remainder of this paper will be to formulate and implement a concurrency control semantics that is useful and does not sacrifice scalability.

3. SEMANTICS AND SYSTEM MODEL

In this section, we present an isolation model, Read Atomic isolation, which we will subsequently implement via RAMP transactions, and our system model for partitioned databases. To ensure RAMP transaction scalability, we also formulate a set of strict scalability criteria: synchronization and partition independence.

3.1 RA Isolation

To begin, we formalize the requirement for atomically visible updates As is customary, we define our isolation model according to phenomena that we wish to disallow, or *anomalies* [2]. Informally, to provide atomically isolated reads and writes, each transaction should read from an unchanging snapshot of database state that is aligned across transactional boundaries.

Using the formalism of Adya [2], we reason about arbitrary ordered sequences of reads and writes to arbitrary sets of items—transactions. Each write creates a *version* of an item and we identify versions of each item by *timestamp*. We discuss timestamp assignment in Section 4 but denote a version of item x with timestamp i as x_i, and timestamps induce a total order on versions of each item (this only induces a partial order *across* items).

We say that a transaction T_j exhibits the *fractured reads* anomaly if transaction T_i writes versions x_m and y_n (x possibly but not necessarily equal to y), T_j reads version x_m and version y_k, and $k < n$. As an example, consider the following two transactions, denoting write as w and read as r:

$$T_1 : w(x_1)\ w(y_1)$$
$$T_2 : r(x_1)\ r(y_0)$$

T_2 exhibits fractured reads because T_2 reads x_1 (written by T_1) and y_0 (not written by T_1) but T_2 did not read y_1 (a later version of y written by T_1). In the example above, to prevent fractured reads, T_2 should have read either x_0 and y_0 *or* x_1 and y_1. This anomaly is similar to Adya's "Missing Transaction Updates" definition (from *PL-2+*) [2] applied to as applied to only immediate read dependencies rather than all transitive dependencies.

We say that a system provides *Read Atomic* isolation (RA) if it prevents fractured reads anomalies and also prevents transactions from reading uncommitted, aborted, or intermediate data. RA is simply a restriction on write *visibility*—that is, if the ACID Atomicity property implies that all or none of a transaction's updates are performed, RA requires that all or none of a transaction's updates are made visible to other transactions. RA does not prevent concurrent updates to a given data item (i.e., anti-dependency cycles, or *G2* or *G-SIa* [2]) as in Snapshot Isolation or Serializability but provides transactions with a "snapshot" view of the database that respects transaction boundaries. RA is stronger than *PL-2* (Read Committed), but weaker than *PL-SI*, *PL-CS*, and *PL-2.99* [2].

In this work, we develop algorithms that provide RA for transactions on partitioned databases. We will call these operations Read Atomic Multi-Partition (or *RAMP*) transactions.

3.2 RA Implications and Limitations

We have carefully chosen RA isolation for its semantics, which match our use cases (and which we revisit in Section 5), and for its scalability. If a programmer can identify when RA is necessary and sufficient for her application, the addition of RA isolation enables faster operation without compromising correctness. However, RA is *not* sufficient for all applications, which we discuss here.

Encouraged programming patterns. From a programmer's perspective, we have found RA isolation to be most easily understandable (at least initially) as 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 and should be familiar to users of isolation models like Read Committed and, in the case of Write Skew, Snapshot Isolation, which also surface (respectively, a superset and a subset of) RA's serialization anomalies.

Nevertheless, our choice of semantics is primarily guided by use cases we have described rather than a search for additional, exotic programming paradigms. Section 5 elaborates, but, for secondary indexing, the primary data and secondary index are each items that can be updated and read atomically. Items containing foreign key constraints can be modified atomically together, while derived data can be updated and read atomically with its base data.

Concurrent updates. RA does not prevent concurrent updates. Applications requiring serial access to state (e.g., due to possible integrity constraint violations [28]) should choose a stronger model that—at the expense of scalability—allows mutual exclusion. RA is sufficient for write-only transactions [3] and applications with commutative operations or reconciliation policies (e.g., "last writer wins") [22, 42]. For example, RA is an incorrect choice for an application that wishes to maintain positive bank account balances in the event of withdrawals. RA is a better fit for our "friend" operation because the operation is write-only and correct execution (i.e., inserting both records) is not conditional on concurrent updates.

Transitivity. RA does not (by itself) provide ordering guarantees across transactions. In our RAMP transactions, we will provide PRAM consistency, whereby each user's writes to each item are serialized [31] (i.e., "session" ordering [20]), and once a user's operation completes, all other users will observe its effects. This provides transitivity with respect to each user's operation but not across users. For example, if a user updates her privacy settings and subsequently posts a new photo, the photo cannot be read without the privacy setting change [15]. However, PRAM does not respect the *happens-before* relation [4] across users unless users explicitly specify dependencies. If Sam reads Mary's comment and replies to it, other users may read Sam's comment without Mary's comment. RAMP transactions can leverage explicit causality [6] in this case but happens-before is not provided by default. If required, we believe that it is possible to enforce happens-before by sacrificing partition independence (defined below) but, due to scalability concerns (e.g., [6] and Section 6), do not further explore this possibility.

3.3 System Model

Before continuing, we first discuss our model for distribution and scalability. We will consider partitioned databases, whereby a set of items is spread across many servers, or *partitions*, such that no one server contains the entire set of items. Users submit transactions to servers, where they are executed. Transactions either *commit*, signaling success, or *abort*, signaling failure. For our initial presentation, we will consider *linearizable* servers such that there is only one (logical) copy of each data item [4]. In our examples, all data items have the null value, $\perp$, at database initialization and all transactions commit.

Scalability criteria. As we hinted in Section 1, large-scale deployments often eschew transactional functionality on the assumption that it is too expensive or unstable in the presence of failure and degraded operating modes [9, 11, 13, 15, 22, 25, 26, 39, 44]. Our goal in this paper is to provide robust and scalable transactional functionality, and, so we first define criteria for "scalability":

Synchronization independence ensures that one client's transactions cannot cause another client's to block and that each client's transaction eventually commits or aborts itself. This prevents one transaction from causing another to abort—which is particularly

important in the presence of partial failures—and guarantees that each client is able to make useful progress. In the absence of failures, this maximizes useful concurrency. In the distributed systems literature, synchronization independence for replicated transactions has been referred to as *transactional availability* [5].

While many applications can limit their data accesses to a single partition via explicit data modeling [7, 19, 25, 39] or planning [18, 36], this is not always possible. In the case of secondary indexing, there is a tangible cost associated with requiring single-partition updates (scatter-gather reads), while, in social networks like Facebook and large-scale hierarchical access patterns as in Rainbird, perfect partitioning of data accesses is near-impossible. Accordingly:

Partition independence ensures that, in order to execute a transaction, a client never has to contact partitions that its transaction does not access. This prevents failure of partitions unrelated to a transaction from affecting the transaction's outcome. In the absence of failures, this minimizes synchronous communication between partitions. In the distributed systems literature, partition independence for replicated data is called as *replica availability* [5, 23].

In addition to the above requirements, we will seek to limit the *metadata overhead* of algorithms. There are many potential solutions for providing atomically isolated transactions with synchronization and partition independence that rely on storing prohibitive amounts of state. As a straw-man solution, each transaction could send each of its writes to each item's respective partition, but this requires quadratic storage and is likely infeasible for most workloads. Similarly, as we discuss in Section 7, a class of existing solutions requires metadata proportional to the number of servers in the cluster or, worse, the number of keys in the database. We will attempt to minimize this *metadata*—that is, data that the transaction did not itself write but which is required for correct execution. In our solutions, we will specifically provide constant-factor metadata overheads (RAMP-S, RAMP-B) with an optional overhead that is linear in transaction size (but independent of value size; RAMP-L).

4. RAMP ALGORITHMS

Given specifications for scalability and RAMP semantics, we proceed to develop algorithms for achieving both. For simplicity, we first focus on providing read-only and write-only transactions with a "last writer wins" conflict reconciliation policy, then subsequently describe how to perform read/write transactions.

At a high level, RAMP transactions allow reads and writes to proceed concurrently. This introduces a race condition: a transaction's reads might only reflect a portion of another transaction's writes (i.e., fractured reads might occur). RAMP readers detect this race using metadata attached to writes and fetch the missing writes from their respective partitions (which employ multi-versioning). Writers use a two-phase protocol and only make their writes visible once they are present on all respective partitions. This ensures that readers never wait for writes that have not yet arrived at a partition. We develop three algorithms that provide a trade-off between the amount of metadata required and the expected number of extra reads to fetch missing writes. If techniques like locking [8] couple mutual exclusion with visibility of writes, RAMP transactions control visibility but admit concurrent execution.

4.1 RAMP-List

To begin, we consider a RAMP algorithm that is based on storing metadata in the form of lists of items, called RAMP-L (Algorithm 1). The algorithm is prototypical of others that we will introduce.

Overview. Each client's writes in RAMP-L (lines 14–21) are augmented with a timestamp (line 15; used for identifying versions and in "last writer wins" reconciliation) and list of items written in the transaction (line 16; to be used by readers, below). We call the set of versions created in a given transaction *sibling versions* and the set of items that were written transaction *sibling items*. We specifically discuss timestamp generation in Section 4.4, but a unique client ID or a hash of the transaction contents coupled with a client-local transaction sequence number suffices.

Writes proceed in two phases: a first round of communication sends each written version to its respective partition (a PREPARE phase, wherein each server places its versions into a set, *versions*; lines 1, 17–19), and a second round marks versions as committed (a COMMIT phase, wherein each server updates *lastCommit*, a mapping that contains the highest timestamp over committed transactions on each item; lines 2, 20–21). Timestamps are used to determine a total order on committed transactions for the purposes of choosing a "winning" write but do not otherwise represent any other sequence or ordering of committed transactions: for example, $lastCommit[k] = 2$ does not imply that a transaction with timestamp 1 has committed or even that such a transaction exists.

Client read transactions (lines 22–33) proceed in two rounds: the first round (lines 23–30) fetches the highest-timestamped committed version for each item (as recorded in *lastCommit*; lines 10–11, 25). Using the returned list of sibling items, clients calculate whether the first round of versions is missing any versions (lines 26–29). If so, clients issue a second round of reads to specifically request the required versions (lines 30–32) from their respective partitions (line 13). Possibly after one round and definitely after two rounds, readers will have assembled a transactionally atomic set of versions to return (line 33).

Figure 1 depicts an execution of two RAMP-L transactions on two partitions; T_2 fetches y_1 explicitly from P_y because T_1 has issued COMMIT on P_x but not (yet) on P_y.

Correctness. To prove RAMP-L provides RA isolation, we show that the two-round read protocol returns a transactionally atomic set of versions. To do so, we will formalize criteria for atomic sets of versions in the form of *companion sets*.

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

Claim 1 (Companion sets are atomic)**.** A companion set does not contain fractured reads.

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

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

Theorem 1. RAMP-L provides Read Atomic isolation.

Proof 1. Each write in RAMP-L contains information regarding its siblings, which can be identified by item and timestamp. Given a

Algorithm 1 RAMP-L

Server-side Data Structures

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

Server-side Methods

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

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

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

Client-side Methods

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

22: **procedure** GET_ALL(I : set of items)
23: $resp \leftarrow \{\}$
24: **for** $i \in I$ **do**
25: $resp[i] \leftarrow$ GET($i, ts_{req} = \perp$))
26: $v_{latest} = \{\}$
27: **for** response $r \in resp$ **do**
28: **for** $i_{tx} \in r.md$ **do**
29: $v_{latest}[i_{tx}]$.updateIfGreater($r.ts_v$)
30: **for** item $i \in I$ **do**
31: **if** $v_{latest}[i] > resp[i].ts_v$ **then**
32: $resp[i] \leftarrow$ GET($i, ts_{req} = v_{latest}[i]$))
33: **return** $resp$

set of RAMP-L 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-L 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-L 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-L never reads aborted or non-final (intermediate) writes. This establishes that RAMP-L provides RA.

Scalability and independence. RAMP-L also provides the independence guarantees from Section 3.3. The following invariant over *lastCommit* is core to RAMP-L 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.

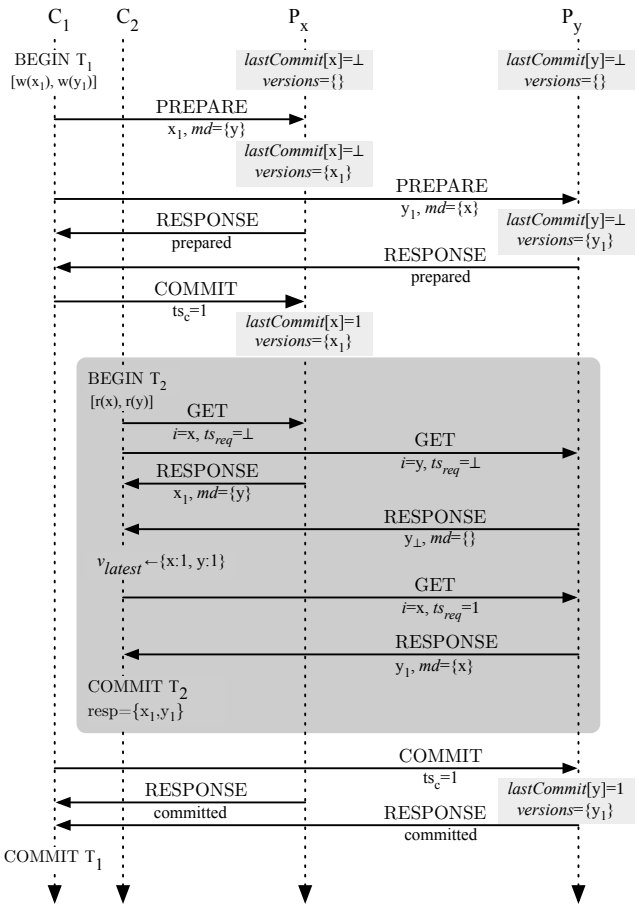

Figure 1: Space-time diagram for RAMP-L execution for T_1 and T_2 (from Section 3) performed by clients C_1 and C_2 on partitions P_x and P_y. Because T_1 overlaps with T_2, T_2 must perform a second round of reads to repair the fractured read between x and y. T_1's writes are assigned timestamp 1. Lightly-shaded boxes represent current partition state (*lastCommit* and *versions*), while the single darkly-shaded box encapsulates all messages exchanged during C_2's execution of transaction T_2.

Invariant 1 is maintained by RAMP-L'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*).

Theorem 2. RAMP-L provides synchronization independence.

Proof 2. Clients in RAMP-L do not communicate or coordinate with one another and only contact servers. Accordingly, to show that RAMP-L provides synchronization independence, it suffices to show that server-side operations always terminate. PREPARE and COMMIT methods only access data stored on the local partition and do not block due to external coordination or other method invocations; therefore, they complete. GET requests issued in the first round of reads have $ts_{req} = \perp$ and therefore will return the version corresponding to $lastCommit[k]$, which was placed into *versions* in a previously completed PREPARE round. GET requests issued in the second round of client reads have ts_{req} set to the client's calculated $v_{latest}[k]$. $v_{latest}[k]$ is a sibling of a version returned from *lastCommit* in the first round, so, due to Invariant 1, the requested

Algorithm 2 RAMP-S

Server-side Data Structures
same as in RAMP-L (Algorithm 1)

Server-side Methods
PREPARE, COMMIT same as in RAMP-L

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

Client-side Methods

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

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-L operations do not depend on the success or failure of other clients' RAMP-L operations.

While space limitations preclude full proofs of the following claims, they follow directly from the algorithm specification:

Claim 2 (RAMP-L provides partition independence)**.** RAMP-L transactions do not access partitions that are unrelated to each transaction's specified data items and server do not contact other servers in order to provide a safe response for operations.

Claim 3 (RAMP-L RTTs)**.** RAMP-L requires 2 RTTs for writes, one RTT in the absence of concurrent write transactions, and two RTTs during concurrent write transactions.

Claim 4 (RAMP-L metadata requirements)**.** RAMP-L versions contain a timestamp and metadata proportional to the number of each version's sibling items. Each GET request contains one timestamp.

4.2 RAMP-Stamp

RAMP-S trades RAMP-L metadata requirements (linear in the size of a write transaction) for an increase in communication rounds for reads (Algorithm 2). RAMP-S proceeds similarly to RAMP-L, but, instead of storing the entire list of siblings along with each version, RAMP-S only stores the transaction's timestamp (line 18). Upon read, clients first fetch the highest committed timestamp for each item from the respective partition (lines 3, 9–11). However, unlike RAMP-L, clients do not attempt to ascertain the appropriate version to read in a second round. Instead, RAMP-S clients defer this task to each item's partition: RAMP-S clients issue a second round of reads, passing every partition the list of timestamps that they read in the first round (line 14) as a parameter of their second round request (lines 13–14; ts_{list} identifies the companion set). Given this list of timestamps, each server returns the highest-timestamped version in *versions* that matches one of the first-round timestamps (lines 5–6). Unlike RAMP-L, there is no requirement to actually return the value of the version in the first round (returning the version, $lastCommitted[k]$, suffices; line 3).

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

Theorem 3. RAMP-S provides Read Atomic isolation.

Proof 3. To show that RAMP-S provides RA, it suffices to show that RAMP-S second-round reads (*resp*) are a companion set. Given two versions $x_i, y_j \in resp$ such that $x \neq y$, if x is a sibling item of y_j, then x_i must be a companion to y_j. If x_i were not a companion to y_j, then it would imply that x is not a sibling item of y_j (so we are done) or that $j > i$. If $j > i$, then, due to Invariant 1 (which also holds for RAMP-S writes due to identical write protocols), y_j's sibling is present in *versions* on the partition for x and would have been returned by the server (line 6), a contradiction. Each second-round GET request returns only one version, so we are done.

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

Claim 5 (RAMP-S metadata requirements)**.** RAMP-S versions contain a single timestamp. Each GET request is contains a list of timestamps proportional to the number of items in the transaction.

4.3 RAMP-Bloom

RAMP-B (Algorithm 3) strikes a compromise between RAMP-L and RAMP-S. Instead of storing an entire list of siblings as in RAMP-L, clients in RAMP-B store a Bloom filter [10] representing the siblings for each transaction (line 1). Accordingly, RAMP-B proceeds like RAMP-L with a first round of reads from *lastCommitted* (lines 3–5) and clients subsequently compute a list of potential sibling timestamps for each item (lines 7–10; $v_{fetch} \cup resp$ is a companion set). Any potentially missing siblings are fetched in a second round of reads (lines 14). Without false positives, v_{fetch} will be computed (line 10) identically to v_{latest} in RAMP-L (Algorithm 1, line 29). With all false positives, v_{fetch} will contain timestamps for all versions that were returned in the first round (equivalent to ts_{resp} in RAMP-S (Algorithm 2, line 11). Accordingly, RAMP-B allows direct control over the trade-off between metadata size and the expected number of read rounds.

Correctness. The probabilistic nature of the Bloom filter admits false positives. However, given unique transaction timestamps (Section 4.4), requesting false siblings by timestamp and item does not affect correctness:

Theorem 4. RAMP-B provides Read Atomic isolation.

Proof 4. To show that RAMP-B provides Read Atomic isolation, it suffices to show that any versions requested by RAMP-B second-round reads that would not have been requested by RAMP-L second-round reads (call this set v_{false}) do not compromise the validity of RAMP-B's returned companion set. Any versions in v_{false} do not exist: timestamps are unique, so, for each version x_i, there are no versions x_j of non-sibling items with the same timestamp as x_i (i.e., where $i = j$). Therefore, requesting versions in v_{false} do not change the set of results collected in the second round.

Scalability and independence. RAMP-B provides synchronization independence and partition independence. We omit full proofs, which closely resemble those of RAMP-L. The only significant difference from RAMP-L is that second-round GET requests may return $\perp$, but, as we showed in Proof 4, these responses correspond to false positives and therefore do not affect correctness.

Claim 6 (RAMP-B metadata requirements)**.** RAMP-B versions store a

Algorithm 3 RAMP-B

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

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

Client-side Methods

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

single timestamp and a Bloom filter. Each GET request is accompanied by a list of timestamps proportional to the number of false positives in the sibling Bloom filter.

The number of unnecessary second round reads (i.e., which would not be performed in RAMP-L) is controlled by the false positive rate. The rate of false positives is determined by the size of the Bloom filter, which is a constant, configurable overhead per write.

4.4 Additional Details

In this section, we discuss garbage collection of older versions, read-write transactions, fault tolerance, timestamp assignment, and concurrent updates and finally summarize our algorithms.

Garbage collection. As is common in multi-versioned databases, each partition's set of *versions* will grow without bound if old versions are not removed. In RAMP transactions, if a committed version of item k has a lower timestamp than $lastCommit[k]$, the version 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). Partitions can detect this condition by gossiping the timestamps of items that have been overwritten and have not been returned in the first round of any ongoing read transactions. However, in practice, it is easier to set a wall-clock-based expiration time on overwritten, committed versions and simply limit the running time of read transactions [33, 34]. Under this latter mechanism, the maximum number of versions 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.

Read-write transactions. Until now, we have focused on read-only and write-only transactions. As discussed in Section 3.2, read-write functionality is often desirable, so, we can extend our algorithms to provide read-write transactions. As a simple solution, if transactions pre-declare the data items they wish to read, then the client can execute a GET_ALL transaction at the start of execution to pre-fetch all items; subsequent accesses to those items can be served from this pre-fetched set. Clients can buffer any writes and, upon transaction commit, send all new versions to servers (in parallel) via a PUT_ALL request. Again, this may result in anomalies due to concurrent update but does not violate RA isolation. Given the benefits of pre-declared read/write sets [18, 36, 43] and write buffering [17, 41], we believe this is a reasonable strategy.

Alg.	RTTs/operation			Metadata (+stamp)	
	W	R (stable)	R (O)	Stored	Per-Request
RAMP-L	2 (1)	1	2	txn items	-
RAMP-S	2 (1)	2	2	-	stamp/item
RAMP-B	2 (1)	$1+\varepsilon$	2	Bloom filter	stamp/item

Table 1: Comparison of algorithms: RTTs required for writes (W; always 2, unless session guarantees are dropped, then 1), reads (R) during quiescence (no concurrent writes) and in the worst case (O), stored metadata and metadata attached to read requests (in addition to a timestamp for each).

Timestamps. Timestamps should be unique to a given transaction, and, for PRAM consistency, increase with each client's write. With unique client IDs, combining a client ID and sequence number provide unique transaction timestamps without coordination. Without unique client IDs, servers can assign timestamps based on UUIDs or a hash of transaction contents and a local sequence number.

Overwrites. In our algorithms, we have depicted a policy in which versions are overwritten according to a highest-timestamp-wins policy. In practice, and, for commutative updates, users may wish to employ a different policy upon COMMIT: for example, perform set union. In this case, $lastCommit[k]$ contains an abstract data type (e.g., set of versions) that can be updated with a *merge* operation [22, 42] (instead of $updateIfGreater$) upon commit. This treats each committed record as a set of versions, requiring additional metadata (that can be GCed as in Section 4.6; cf. Section 5).

Summary: Communication versus Metadata. We summarize the trade-off between communication and stored metadata in Table 1. RAMP-L requires metadata linear in transaction size, while RAMP-S and RAMP-B require constant metadata at the expense of increased read RTTs compared to RAMP-L when reads to not overlap with writes to the same items. However, when writes *do* overlap, in the worst case, all algorithms require two RTTs for reads.

4.5 Distribution and Fault Tolerance

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

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

is therefore remarkably similar to eventually consistent systems.

Stalled Operations. Core to the RAMP write algorithms is a two-phase commit protocol similar to atomic commitment protocols that have been studied extensively in the literature [8]. Notoriously, in the event of communication or total failures, every atomic commitment protocol may cause processes to become blocked [8]. Fortunately, due to synchronization independence, a blocked RAMP write (that is, incomplete PUT_ALL operations, due to either failed clients, failed servers, or network partitions) cannot cause other writers to become blocked. Accordingly, blocked writes will not affect correctness under RA, but blocked writes will effectively act as resource leaks on partitions: individual servers will retain versions indefinitely unless action is taken (e.g., it is unsafe to simply discard these versions after a time-out). As an additional concern, it is desirable if writes that block while sending COMMIT commands eventually become visible on all partitions (guaranteeing "eventual consistency," or convergence across partitions). CTP provides liveness in the form of convergence and possible removal of stalled writes but does not affect the RA safety guarantees discussed in earlier sections.

There are several strategies for addressing blocked writes, but we adopt the Cooperative Termination Protocol (CTP) described in [8]. CTP allows RAMP servers to unblock writes in the case that a client fails before every server receives PREPARE and in the case that a client fails before delivering all (but at least one) COMMIT command. This leaves a window of vulnerability between the time that every server performs PREPARE and any server performs COMMIT; in this case, versions will potentially be retained unnecessarily (but RA and convergence will be guaranteed).

Informally, in the CTP algorithm, 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 sibling's servers. If a sibling server S_c has received COMMIT for T, then S_p can COMMIT T. If a sibling server S_a has not received PREPARE for T, S_a and S_p can promise never to PREPARE or COMMIT T in the future and S_p can safely discard its versions. A client recovering from a failure can read from the servers to determine if they unblocked its write.

The CTP algorithm (which we evaluate in Section 6) has the benefit that it only runs when writes block (or time-outs fire), and runs *asynchronously* with respect to non-blocked operations. CTP requires that PREPARE messages contain a list of servers involved in the transaction (a subset of RAMP-L metadata but a superset of RAMP-B and RAMP-S) and that servers remember when they COMMIT and "abort" writes (e.g., in a log file). (A alternative RAMP-L implementation can simply check *lastCommit* instead of retaining COMMIT decisions.) This is a modest overhead, but, as a necessary consequence of the above impossibility result, CTP may not unblock the write in all scenarios (resulting in wasted, asynchronous communication) and may cause the write to fail if client writes are delayed greater than the time-out. We found CTP most attractive due to its limited impact on normal-case performance (versus, say, synchronously replicating the coordinator [24]) and our experience thus far has validated this decision.

4.6 Further Optimizations

Our RAMP algorithms have several possible optimizations:

Faster commit detection. In our algorithms, if a server returns a value in response to a GET with a timestamp for transaction T greater than the current value of *lastCommit*, T must have been committed on at least one partition, so the server can update *lastCommit* with T's timestamp. 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-L and RAMP-B RTTs).

Metadata garbage collection. Once all of transaction T's writes are reflected in their respective partitions' *lastCommit*, readers are guaranteed to read T's writes (or later writes). Therefore, non-timestamp metadata for T's writes stored in RAMP-L and RAMP-B (sibling lists and Bloom filters) can therefore be discarded. Detecting that all servers have performed COMMIT can be performed asynchronously via a third round of communication performed by either clients or servers.

One-phase writes. We have considered two-phase writes, but, if a user does not wish to read her writes (thereby sacrificing PRAM guarantees outlined in Section 3.2), the client can return after issuing its PREPARE round (without sacrificing durability). The client can subsequently execute the COMMIT phase asynchronously, or, similar to optimizations presented in Paxos Commit [24], the servers can exchange PREPARE acknowledgements with one another and decide to COMMIT autonomously. This optimization safe because multiple PREPARE phases can overlap under RA.

5. INTERLUDE: APPLYING RAMP

Before experimentally evaluating RAMP performance, we revisit our target use cases from Section 2 in light of our RA isolation and the RAMP algorithms. To be explicit, the contribution of this section is a demonstration of how RAMP can be applied to the problem of concurrency control over these data structures, not in the actual data structures themselves.

Multi-entity update. Our initial presentation of the RAMP algorithms was targeted towards multi-entity update: if users wish to update two or more items and make the effects of their updates visible at once, they can package the updates using a RAMP transaction and invoke the PUT_ALL function from any of the three algorithms. A user creating a profile and adding a profile picture to a Bulletin Board application could use a RAMP transaction.

Foreign key constraints. When inserting new associations, users can perform a RAMP transaction and include entities from both edges of the association in their operation. In the Tao example, each bi-directional link creation becomes a RAMP PUT_ALL call. When deleting associations, users can "tombstone" (i.e., delete any entries with associations via a special record that signifies deletion) [45] the opposite end of the association to avoid dangling pointers.

Secondary indexing. The secondary index entry for a given attribute can be represented by an unordered set of version IDs that match the attribute (e.g., a list of IDs of people with blue hair). Insertions can be modeled as additions to the corresponding entry set, deletions as removals, and updates as a "tombstone" deletion from one set (that can, if desired, be asynchronously garbage collected [40, 45]) and an insertion into another.

Derived data. RAMP transactions can be used to ensure the atomicity of updates to base and derived data. In the case of select-project views, a simple solution is to treat the derived data as a separate table and perform maintenance as needed: upon COMMIT, new rows can be inserted/deleted according to the view specification, and, during reads from the prepared state, the view can be computed on demand (i.e., merge the previously read *lastCommit* state with the *prepare*d delta)—effectively, lazy view maintenance [46]. From the perspective of RAMP transactions, this treats the derived data record as a set of records, with metadata for each record in the base table. We can generalize this strategy by maintaining a full copy of the base data at the derived data partition (thus avoid-

ing the "state bug" [14]), but, for larger, and more complex derived data, such as recursive views, this solution becomes inefficient [14] (and is not always necessary [27]). We believe it is possible to perform this more complex maintenance, but the solution loses some of its simplicity. For example, to avoid Lost Update anomalies, we can implement derived counters with commutative data structures like the CRDT G-Counter (recently adopted by Riak), which represents the counter by a vector clock, with one entry per incrementing process, and which can be merged (and garbage collected) via well-defined protocols (PN-Counters are sufficient for increment/decrement) [40]. We view RAMP's contribution as providing a concurrency control mechanism for maintaining read atomicity between derived and primary data but reserve the physical maintenance of more complex views (i.e., beyond those discussed in [37, 39, 44]) as related (and future) work.

6. EVALUATION

Given our algorithms and analysis, we proceed to experimentally demonstrate our RAMP algorithms' scalability as compared to existing transactional and non-transactional mechanisms. RAMP-L, RAMP-B, 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 under contention and scales linearly to over 7.4 million operations per second on 100 servers. These experimental outcomes validate our choice to pursue synchronization- and partition-independent mechanisms for RAMP transactions.

6.1 Experimental Setup

To demonstrate the effect of concurrency control on performance and scalability, we implemented several concurrency control algorithms in a partitioned, multi-versioned, main-memory database similar to H-Store [30]. Servers are arranged as a distributed hash table with partition placement determined by a random hash. As in stores like Dynamo [22], users 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-L, RAMP-S, and RAMP-B and employ time-based garbage collection configured to 5 seconds as described in Section 4.4. RAMP-B uses the Hadoop MurmurHash2.0 implementation with 256 bits and four hashes per item; we did not change these parameters across experiments to demonstrate the effects of filter saturation. We did not implement the latter two optimizations suggested in Section 4.4 because we believe that most users desire PRAM guarantees and metadata overheads were negligible.

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.

We also consider three lock-based mechanisms: long write locks and long read locks, providing Repeatable Read isolation (*PL-2.99*; denoted LWLR), long write locks with short read locks, providing Read Committed isolation (*PL-2L*; denoted LWSR; does not provide RA), and long write locks with no read locks, providing Read Uncommitted isolation [2] (LWNR; also does not provide RA). While only LWLR provides RA, LWSR and LWNR provide a useful basis for comparison, particularly in measuring concurrency-related locking overheads (i.e., LWNR is a lower bound for the cost of avoiding Lost Update for writes, as in Snapshot Isolation). To avoid deadlocks, the system totally 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 a variant of dynamic partitioning where, for each transaction, designated coordinator servers enforce RA isolation (denoted MSTR). We adopt the Eiger system's transactions [34], effectively a lightweight (non-serializable) implementation of Chan and Gray's Read-only Transactions [12] and G-Store [19] (Section 7). Writes proceed via prepare and commit rounds, but any reads that arrive at a partition and overlap with a concurrent write to the same item are indirected to a (randomly chosen, per-transaction) "coordinator" partition to determine whether the prepared writes have been committed (i.e., Eiger's 2PC-CI). Writes require two RTTs, while reads require one RTT during quiescence and two RTTs (to a variable number of servers) in the presence of concurrent updates. This effectively provides $PL-2L$ with Cut Isolation (with synchronization but *not* partition independence) [1, 2]. We optimize Eiger's read protocol by having clients determine a read timestamp for each transaction, eliminating an extra RTT.

We believe that this range of lock-based strategies (LWNR, LWSR, LWNR), recent comparable approach (MSTR), and best-case (NWNR; no concurrency control) baseline provides a reasonable spectrum of strategies for comparison.

Environment and benchmark. We evaluate each algorithm under YCSB [16] and deploy variable-sized sets of servers on public cloud infrastructure. We employ `cr1.8xlarge` instances on Amazon EC2 and, by default, consider five partitions on five servers. We group sets of reads and sets of writes into read-only and write-only transactions (default size: 4 operations), and, by default, use the default YCSB workload (`workloada`, with Zipfian-distributed item access) but with a 95% read and 5% write proportion, reflecting read-mostly applications (Section 2, [11,34,44]; e.g., Tao's 500 to 1 reads-to-writes [11,34], Espresso's 1000 to 1 Mailbox application [39], and Spanner's 3396 to 1 advertising application [17]).

By default, we use 5000 concurrent clients split across 5 separate EC2 instances and, to fully expose our metadata overheads, use a 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, *not* values) of size 11 bytes and a transaction length L, RAMP-L stores $11L$ additional bytes of metadata, while RAMP-B stores a constant 32 bytes. We successively vary several parameters, including database size, read proportion, transaction length, value size, number of partitions, and number of clients and report the average of three sixty-second trials.

6.2 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-L and RAMP-B always outperform lock-based and MSTR techniques, while RAMP-S outperforms lock-based techniques and often outperforms MSTR. We proceed to demonstrate this behavior over a variety of conditions:

Number of Clients. RAMP performance scales well with increased load and incurs little overhead (Figure 2). With few concurrent clients, there are few concurrent updates and therefore few second-round reads; performance for RAMP-L and RAMP-B is close to or even matches that of NWNR. At peak throughput (at 10,000 clients), RAMP-L and RAMP-B pay a throughput overhead of 4.2% compared to NWNR. RAMP-L and RAMP-B exihibit near-identical performance; the RAMP-B Bloom filter triggers few false positives (and therefore few extra RTTs compared to RAMP-L). RAMP-S incurs greater

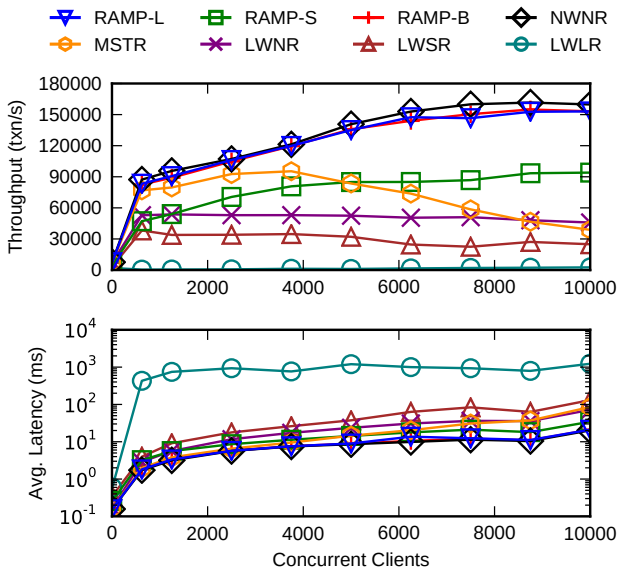

Figure 2: Performance versus number of concurrent clients.

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-L and RAMP-B performance. Ultimately, all four of these algorithms bottleneck on CPU. In all configurations, the algorithms achieve low latency (RAMP-L, RAMP-B, NWNR less than 35ms on average and less than 10 ms at 5,000 clients; RAMP-S less than 53ms, 14.3 ms at 5,000 clients).

In comparison, the other algorithms perform less favorably. In contrast with our RAMP algorithms, MSTR 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 MSTR between RAMP-S and RAMP-B. However, the number of in-flight writes increases with load and therefore the number of MSTR commit checks increases. This in turn decreases throughput, and, with 10,000 concurrent clients, MSTR performs so many commit checks per read (over 20% of reads trigger a commit check, and, on servers with YCSB's Zipfian-distributed "hot items", each commit check requires checks for an average of 9.84 transactions) that it underperforms the LWNR lock-based scheme. Multi-partition locking is expensive, even for cutting-edge locking mechanisms [36]. At 10,000 concurrent clients, the most efficient algorithm, LWNR, attains only 28.6% of the throughput of NWNR, while the least efficient, LWLR, attains only 1.6% (peaking at 3,000 transactions per second).

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

Read Proportion. Writes under RAMP-L and RAMP-B incur two primary costs: first, writes take two RTTs (compared to 1–2 RTTs for reads; Table 1), and, second, increased write activity increases the chance that RAMP-L and RAMP-B reads will race with a write and incur a second RTT. Therefore, increasing the proportion of writes incurs throughput penalties for RAMP-L and RAMP-B: with all write transactions, all RAMP algorithms are equivalent (two RTT) and achieve approximately 65% of the throughput of NWNR. With all reads, RAMP-L, RAMP-S, NWNR, and MSTR are identical, with a single RTT. Between these extremes, RAMP-L and RAMP-S scale near-linearly with the write proportion. In contrast, lock-based protocols fare poorly as contention increases, while MSTR incurs high penalties due to commit checks.

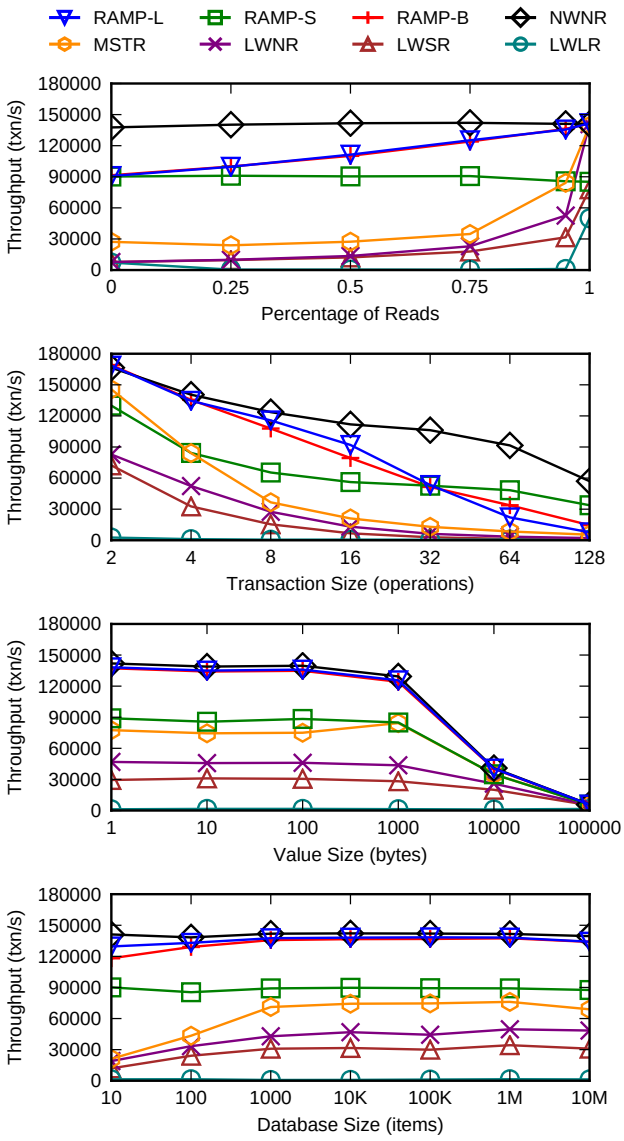

Figure 3: Effect of workload parameters on throughput.

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

Value size. Value size similarly does not seriously impact relative throughput. At a value size of 1B, RAMP-L is within 2.3% of NWNR. However, at a value size of 100KB, RAMP-L is within 0.01% of NWNR: write request rates slow, decreasing concurrent writes (and subsequently second-round RTTs) and the overhead due to metadata. Nonetheless, absolute throughput drops by a factor of 24 as value sizes moves from 1B to 100KB.

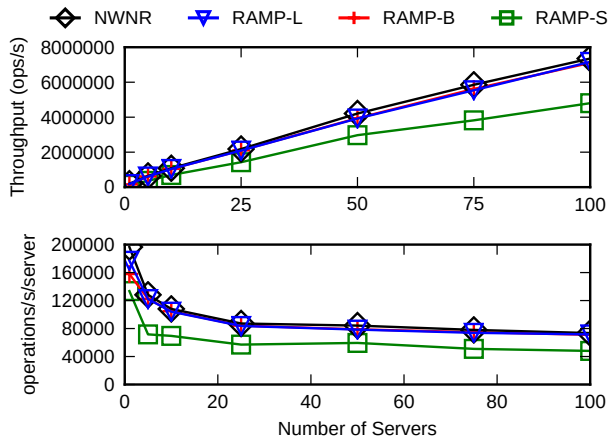

Figure 4: RAMP algorithms scale linearly to over 7 million operations/s with comparable performance to NWNR baseline.

Database size. Unsurprisingly, our RAMP algorithms are robust to high contention for a small set of items: with only 1000 items in the database, RAMP-L achieves throughput within 3.1% of NWNR; with 10 items, within 8.2%. RAMP algorithms are effectively contention-agnostic: with 10 items, each item is overwritten an average of 2,500 times per second, but performance is hardly affected. This behavior is not possible with MSTR because reads are almost always likely to overlap with a write transaction, triggering a indirected commit check, even if servers could have returned an atomic set of writes. RAMP-L and RAMP-B triggered fewer second-round reads than MSTR incurred commit checks.

Overall, RAMP-L and RAMP-B exhibit performance similar 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-L and RAMP-B operations take two RTTs and performance trends towards that of RAMP-S, which provides a constant two-RTT overhead. In contrast, lock-based protocols perform poorly under contention while MSTR triggers more commit checks than RAMP-L and RAMP-B trigger second round reads. The ability to allow clients to independently verify companion sets instead of relying on even decentralized, dynamic coordinator servers (as in MSTR) enables good performance despite a range of adverse conditions (e.g., high contention).

CTP Overhead. We also evaluated the overhead of blocked writes in our implementation of the Cooperative Termination Protocol discussed in Section 4.5. To simulate blocked writes, we artificially dropped a percentage of COMMIT commands in PUT_ALL calls such that clients returned from writes early and partitions were forced to complete the commit via CTP. This behavior is worse than expected because "blocked" clients continue to issue new operations. The table below reports the throughput reduction as the proportion of blocked writes increases (compared to no blocked writes) for a workload of 100% 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—5*s* in our experiments), so a modest failure rate of 1 in 1000 writes has a limited effect. The higher failure rates produce a near-linear throughput reduction but, in practice, a blocking rate of even a few percent is likely indicative of larger systemic failures. As Figure 3 hints, the effect of additional metadata for the participant list in RAMP-B and RAMP-S is limited, and, for our default configuration (fewer writes), we observe similar trends but with throughput degradation of 10% or less across all configurations. This validates our initial motivation behind the choice of CTP: average-case overheads are minimal.

6.3 Experimental Results: Scalability

Having demonstrated our RAMP algorithms' performance with respect to existing algorithms, we now validate our chosen scalability criteria by demonstrating linear scalability of RAMP transactions 100 servers. We deployed an increasing number of servers within the us-west-2 region and, to mitigate the effects of hot items during scaling, configured uniform random access to items. We were unable to include more than 20 instances in an EC2 "placement group," which guarantees 10 GbE connections between instances, so, past 20 servers, servers communicated over a degraded network. Around 40 servers, we exhausted the us-west-2b "availability zone" (datacenter) capacity and had to allocate our instances across the remaining zones, further degrading network performance. However, as shown in Figure 4, each RAMP algorithm scales linearly, even though in expectation, at 100 servers, all but one in 100M transactions is a multi-partition operation. In particular, RAMP-L achieves slightly under 7.2 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-L throughput was always within 10% of NWNR. With 100 servers, RAMP-L was within 2.6%, RAMP-S within 3.4%, and RAMP-S was within 45% of NWNR. In light of our independence criteria, this behavior is unsurprising, and we see few barriers to further scaling.

7. RELATED WORK

Transactions and concurrency control have a long history in the database community, and replicated databases have operated for several decades and designs provide a spectrum of isolation guarantees at varying costs to performance and availability [8].

Serializability. At the strong end of the isolation spectrum is serializability, which provides programmers with "single system image" semantics (and therefore RA). A range of techniques for enforce serializability in distributed databases [8], including tentative update and multi-version concurrency control schemes (e.g. [38]) and hybrid locking schemes (e.g. [32]). These useful semantics come with costs in the form of decreased concurrency (e.g., contention and/or failed optimistic operations) and provably limited availability during partial failure [1, 21]. Many designs [19, 30] exploit cheap serializability within a single partition but face scalability challenges for distributed operations. Recent industrial efforts like F1 [41] and Spanner [17] have improved scalability via aggressive hardware advances but, reported throughput is limited to ~20 and 250 operations per second. Multi-partition serializable transactions remain expensive [18, 29, 36].

Weak isolation. The remainder of the isolation spectrum is more varied. RDBMSs have offered (and often default to) non-serializable isolation models for decades [5, 35]. These "weak isolation" levels allow greater concurrency and fewer system-induced aborts compared to serializable execution but provide weaker semantic guarantees. In recent years, many "NoSQL" designs have avoided cross-partition transactions, effectively providing Read Uncommitted isolation in many industrial databases such PNUTS [15], Dynamo [22], TAO [11], Espresso [39], Rainbird [44], and BigTable [13]. These systems avoid penalties associated with stronger isolation but almost always sacrifice transactional guarantees (and therefore 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 is similar to RAMP-L reads [33]—client read transactions are able to identify causally inconsistent writes and fetch them from servers. However, COPS-GT provides causal consistency (with large metadata during garbage collection stalls) and does not support atomic writes.

Eiger provides write-only transactions [34] by electing a coordinator server to indirect reads during concurrent writes: as a consequence (e.g., Section 6.2) the number of "commit checks" scales linearly with system throughput. This hurts performance and violates partition independence. Dynamic coordinator election is analogous to G-Store's dynamic key grouping [19] but with weaker isolation guarantees; each dynamic master effectively contains as a partitioned completed transaction list (CTL) from [12]'s distributed read-only transactions. Instead of relying on indirection, RAMP transaction clients autonomously assemble RA reads and only require constant factor (or, for RAMP-L, linear in transaction size) metadata compared to Eiger's *PL-2L* (i.e., linear in database size).

RAMP transactions are inspired by a recent proposal for *Monotonic Atomic View* (MAV) transactions: linearly scalable transactions that read from a monotonically advancing view of database state [1]. However, 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). This MAV algorithm is similar to RAMP-L but, as a consequence of its weaker semantics, allows one-round read transactions.

[3] provides non-distributed, serializable write-only transactions via a synchronized, shared counter and serializing read-write transactions are after running write-only transactions. RAMP transactions avoid this synchronization but are not serializable.

While our focus is on the concurrency control aspects of derived data, a large body of literature studies various aspects of maintaining derived data [14, 46], particularly when views can be maintained without retaining base data [27].

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

8. CONCLUSION

This paper described how to achieve atomically isolated multi-partition transactions without incurring the performance and availability penalties of traditional algorithms. We first identified a new isolation level—Read Atomic isolation—that matches the requirements of a large real-world applications. We subsequently described how to achieve RA isolation via scalable, contention-agnostic RAMP transactions. Given that the state of the art relies on inconsistent but fast update, RAMP transactions provide concurrency control for applications requiring secondary indexing, foreign key constraints, and derived data maintenance to maintain performance at scale while providing multi-partition atomicity. By leveraging multi-versioning with a variable but small (and, in two of three algorithms, constant) amount of metadata per write, our RAMP transactions allow clients to detect and assemble atomic sets of versions without coordination and within one and no more than two RTTs with servers. Choosing contention-agnostic 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 a perfect fit for all applications, the many for which they are well-suited will benefit measurably compared to the prior state-of-the-art.

9. REFERENCES

[1] Anonymized for double-blind reviewing.

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

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

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

[5] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. HAT, not CAP: Towards Highly Available Transactions. In *HotOS 2013*.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[27] N. Huyn. Multiple-view self-maintenance in data warehousing environments. In *VLDB 1997*.

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

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

[30] R. Kallman et al. H-Store: a high-performance, distributed main memory transaction processing system. In *VLDB 2008*.

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

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

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

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

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

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

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

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

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

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

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

[42] D. B. Terry et al. Session guarantees for weakly consistent replicated data. In *PDIS 1994*.

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

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

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

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