RAMP-TAO: Layering Atomic Transactions on Facebook's Online TAO Data Store

Audrey Cheng*
UC Berkeley
Berkeley, USA
accheng@berkeley.edu

Xiao Shi
Facebook, Inc.
Cambridge, USA
xshi@fb.com

Lu Pan
Facebook, Inc.
Cambridge, USA
lupan@fb.com

Anthony Simpson
Facebook, Inc.
Cambridge, USA
asimpson96@fb.com

Neil Wheaton
Facebook, Inc.
Cambridge, USA
neilwheaton@fb.com

Shilpa Lawande
Facebook, Inc.
Cambridge, USA
slawande@fb.com

Nathan Bronson*
Rockset
Cambridge, USA
ngbronson@rockset.com

Peter Bailis
Sisu Data
San Francisco, USA
peter@sisudata.com

Natacha Crooks
UC Berkeley
Berkeley, USA
ncrooks@berkeley.edu

Ion Stoica
UC Berkeley
Berkeley, USA
istoica@berkeley.edu

ABSTRACT

Facebook's graph store TAO, like many other distributed data stores, traditionally prioritizes availability, efficiency, and scalability over strong consistency or isolation guarantees to serve its large, read-dominant workloads. As product developers build diverse applications on top of this system, they increasingly seek transactional semantics. However, providing advanced features for select applications while preserving the system's overall reliability and performance is a continual challenge. In this paper, we first characterize developer desires for transactions that have emerged over the years and describe the current failure-atomic (i.e., write) transactions offered by TAO. We then explore how to introduce an intuitive read transaction API. We highlight the need for atomic visibility guarantees in this API with a measurement study on potential anomalies that occur without stronger isolation for reads. Our analysis shows that 1 in 1,500 batched reads reflects partial transactional updates, which complicate the developer experience and lead to unexpected results. In response to our findings, we present the RAMP-TAO protocol, a variation based on the Read Atomic Multi-Partition (RAMP) protocols that can be feasibly deployed in production with minimal overhead while ensuring atomic visibility for a read-optimized workload at scale.

PVLDB Reference Format:
Audrey Cheng, Xiao Shi, Lu Pan, Anthony Simpson, Neil Wheaton, Shilpa Lawande, Nathan Bronson, Peter Bailis, Natacha Crooks, Ion Stoica. RAMP-TAO: Layering Atomic Transactions on Facebook's Online TAO Data Store. PVLDB, 14(12): 3014-3027, 2021.
doi:10.14778/3476311.3476379

*Work done while at Facebook.

Proceedings of the VLDB Endowment, Vol. 14, No. 12 ISSN 2150-8097.
doi:10.14778/3476311.3476379

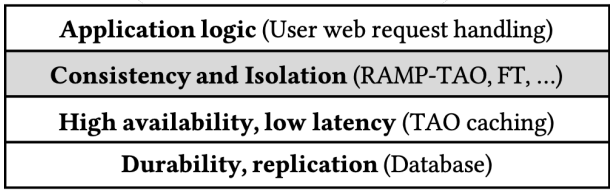


Figure 1: By separating availability and replication concerns from stronger isolation guarantees, we can maintain high performance while ensuring safety properties over data.

1 INTRODUCTION

TAO is a read-optimized, geo-distributed data store that provides online social graph access for diverse product applications and other backend systems at Facebook [21]. Each application-level request can result in hundreds of reads and writes to TAO. In aggregate, TAO serves over ten billion reads and tens of millions of writes per second on a changing data set of many petabytes. Like many other large storage systems [2, 22, 23, 26, 41], TAO prioritizes availability, read latency, and efficiency over strong data consistency and isolation guarantees [45] for its demanding, read-dominant workload.

Originally, TAO focused on simple accesses to nodes and edges in the social graph and provided no transactional APIs, reflecting its goal of supporting a small feature set with high availability at massive scale. However, as applications shifted from directly accessing

TAO to a higher-level query framework, Ent, which makes it easy to express complex operations over the graph, product developers have increasingly desired transactional semantics to avoid having to handle partial failures. In response, TAO engineers implemented failure-atomic write transactions (Section 2).

Similarly, adding support for read transactions would greatly simplify the developer experience by directly enforcing application-level invariants. Currently, under TAO's eventual consistency, a naïve batched query can observe *fractured reads* [17]—reads that capture only part of a write transaction's updates before these updates are fully replicated. As we demonstrate in the first large-scale measurement study of its kind, these anomalies occur 1 out of every 1,500 read batches (Section 3). Given the size of TAO's workload, this relatively modest rate is significant in practice. Moreover, fractured reads are hard to detect in the application layer with an asynchronously replicated system. These anomalies are burdensome for developers to reason about and explicitly handle in order to minimize end user impact.

By providing *atomic visibility* [17], or the guarantee that reads observe either all or none of a transaction's operations, we can introduce a simple and intuitive read transaction API on TAO. However, enabling these semantics presents significant challenges in practice. Due to TAO's scale and storage constraints, we want to avoid coordination and minimize memory overhead. Our implementation of atomic visibility must also be cache-friendly, hot-spot tolerant, and extensible to different data stores. Moreover, we should only incur overhead for applications that opt in (rather than requiring every application to pay a performance penalty). Although we focus on TAO in this work, these challenges apply to many other large-scale, read-optimized systems [2, 41, 49].

In this paper, we introduce a new RAMP-TAO protocol (Section 4), which layers atomic visibility on top of TAO while achieving our performance goals above. While our work is inspired by the Read Atomic Multi-Partition (RAMP) protocols [17], we address several of their drawbacks. The original RAMP transactions impose atomic visibility verification overhead on all reads, require substantial metadata, and assume full support for multiversioning (which TAO lacks). RAMP-TAO leverages the key insight that we only need to guard against fractured reads for recent, transactionally-updated data to reduce the overheads of ensuring atomic visibility.

Our layering strategy takes the "bolt-on" [18] approach to stronger transactional guarantees (Figure 1). We prevent transactions from interfering with TAO's availability, durability, and replication processes, retaining the reliability and efficiency of the system. Only applications that need stronger guarantees incur the resulting performance costs. Furthermore, our protocol exploits existing cache infrastructure and requires minimal changes to TAO internals. Thus, RAMP-TAO is effective for both providing default guarantees across data stores and as a retroactive optimization for massive, read-optimized systems, many of which have sought to strengthen their isolation models [1, 4, 5]. We also describe an optimization of our protocol for bidirectional associations (paired edges in the graph), which represent a special case of failure-atomic transactions. These data structures are ubiquitous in TAO, so any protocol providing atomic visibility needs to be especially efficient for them.

We demonstrate that RAMP-TAO is feasible for production use by benchmarking its latency and memory overheads (Section 5).

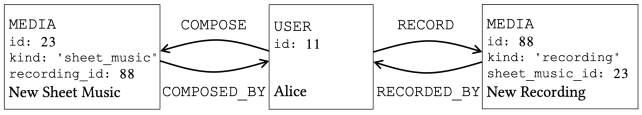


Figure 2: Subgraph for a hypothetical example.

Our prototype implementation provides atomic visibility in a read-optimized environment with one round trip for greater than 99.93% of reads and a modest 0.42% increase in memory overhead.

In summary, we make the following contributions in this paper:

- We report on developer challenges and needs for transactional semantics within Facebook's social graph serving ecosystem (Section 2).
- We present a quantitative study of atomic visibility violations derived from production data to demonstrate the importance of providing this guarantee in a read transaction API.
- We describe a novel RAMP-TAO protocol (Section 4) to efficiently provide atomic visibility for an eventually consistent, read-optimized system.
- We demonstrate the production feasibility of RAMP-TAO by showing it incurs minimal overhead and requires only one round trip for the vast majority of reads (Section 5).

2 OVERVIEW AND MOTIVATION

TAO provides online access to the social graph at Facebook [21]. It is implemented using two layers of graph-aware caches that mediate access to the statically-sharded MySQL database [35]. Updates are replicated asynchronously via the Wormhole pub-sub system [44].

TAO prioritizes low-latency, scalable operations and thus opts for weaker consistency and isolation models to serve its demanding, read-dominant workloads. TAO provides point get, range, and count queries, as well as operations to create, update, and delete objects (nodes) and associations (edges). Its simple graph API is conducive to maintaining high reliability and is sufficient for the vast majority of applications at Facebook. As new applications emerge and as Ent, our higher-layer query framework, evolves, we have been exploring, designing, and implementing additional features such as transactions while preserving TAO's reliability and efficiency.

In this section, we explain the types of transactional guarantees developers desire and highlight corner cases they need to handle before system-level options are offered. We then describe the current approaches to providing failure atomicity. Finally, we demonstrate the importance of atomic visibility for an intuitive read transaction API and considerations for providing stronger guarantees at scale.

2.1 An example

Consider a hypothetical social media product built on top of TAO, with user nodes, media nodes, edges when a user has composed a piece of sheet music, and edges when a user has recorded a song. This product enables musicians to share their sheet music and corresponding recordings together. Let us say that Alice wants to share a piece of music she has composed and recorded so that others can view the sheet music while listening to the recording. The application writes the following edges together (the resulting subgraph is shown in Figure 2):

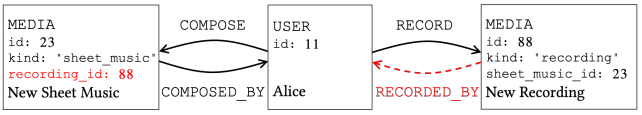


Figure 3: Example of a partial failure. The recording does not refer to its user.

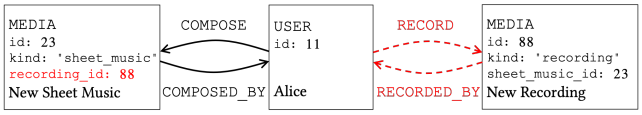


Figure 4: Example of a fractured read. Only the sheet music can be viewed but not the corresponding recording.

$$\texttt{WriteEdge}(\langle 11, \textit{COMPOSE}, 23\rangle)$$
$$\texttt{WriteEdge}(\langle 23, \textit{COMPOSED\_BY}, 11\rangle)$$
$$\texttt{WriteEdge}(\langle 11, \textit{RECORD}, 88\rangle)$$
$$\texttt{WriteEdge}(\langle 88, \textit{RECORDED\_BY}, 11\rangle)$$

Without any transactional guarantees, it is possible that some of the four writes above do not successfully complete, in which case the graph would be left in a confusing state for the application to manually clean up (Figure 3).

Now, suppose the application eventually writes the above edges successfully in a transaction. While this process occurs, let us say that Bob views Alice's new music. The application tries to read edges $\langle 11, \textit{COMPOSE}, \_\rangle$ and $\langle 11, \textit{RECORD}, \_\rangle$. Under TAO's eventual consistency, it is possible that Bob sees the sheet music for the new piece (i.e., the edge $\langle 11, \textit{COMPOSE}, 23\rangle$) without seeing the recording ($\langle 11, \textit{RECORD}, 88\rangle$). This anomaly is a *fractured read*, or a read result that captures partial transactional updates (Figure 4). Handling fractured reads would add complexity to the application logic.

This example demonstrates that we desire two types of transactional guarantees for TAO. First, we need *failure atomicity*, or write atomicity: either all or none of the items in a write transaction are persisted. Without it, the application is left responsible for cleaning up Alice's updates in the database in the case of a failure. Since the application layer lacks full visibility into the underlying system, remediation efforts are often hampered by scaling limits and corner cases (e.g., a failed write is indistinguishable from a write that is not yet replicated).

Second, Bob needs to see either both Alice's sheet music and recording or none of them. We desire *atomic visibility*, a property that guarantees that either all or none of any transaction's updates are visible to other transactions. If we do not provide atomic visibility, developers have to reason about all the possible intermediate states the application might observe and then hide them in the product or determine they can be revealed to users. These application-level workarounds are a source of unnecessary complexity and increase the cognitive load of developers.

We emphasize the distinction between failure atomicity and atomic visibility, particularly for our system. The prevalence of caching and asynchronous replication in TAO lengthens the period before the effects of a failure-atomic transaction can be globally observed.

The remainder of this section describes TAO's current transactional APIs for failure atomicity and why scalability is a prerequisite for any approach to providing atomic visibility.

2.2 Current approaches for failure atomicity

Over the years, TAO has developed failure-atomic transactions APIs in direct response to product engineer needs. Depending on the properties of a transaction, such as what items are involved and where they are physically located, TAO executes the transaction with varying strategies.

Single-shard MultiWrites. The first transactional API TAO added was the `MultiWrite`—a multi-put API restricted to a single shard. MultiWrites provide failure atomicity by leveraging the underlying MySQL transactions and their ACID properties [6].

Over time, applications have run into several scaling limits with the single-shard MultiWrite API. To start, developers must have the foresight to colocate relevant data to use this API—opting in late in their development cycle requires laborious data migration or schema redesign. Moreover, a growing application may hit the physical limit of the underlying database machines. Colocation for MultiWrites can lead to shard capacity imbalance, resulting in abnormally large, hot, or under-utilized shards. When physical machines are no longer sufficient, developers need to overhaul the application and physical data schema, a burdensome and time-consuming process. Furthermore, sharding is one of TAO's fundamental strategies for horizontal scalability. Colocating all application data on a single shard defeats the purpose of this capability. Finally, applications may not have exclusive data ownership, so colocating data is not always feasible. As such, developers need a more powerful API that is not constrained by shard boundaries.

Cross-shard transactions. The limitations of single-shard transactions led application developers to design ad-hoc solutions for modifying data atomically across shards. However, these workarounds often faced issues because applications did not have full visibility into TAO and its underlying database. For example, product engineers could not easily distinguish between a non-existent write and a write that was not yet replicated. As a result, it was hard for developers to make data management decisions at the application layer.

TAO developers realized that product engineers needed a more general solution for cross-shard transactions at the infrastructure level. Such an API would reduce redundant efforts by different teams, support a centralized framework for simplified monitoring and debugging, and allow the complex issues of protocol corner-cases, scalability, and data clean-up after failures to be addressed by system experts rather than product developers.

Consequently, TAO added failure-atomic, cross-shard write transactions, which use a two-phase commit protocol (2PC) [27] with persisted progress for failure recovery. Each transaction is assigned a unique id and uses two-phase locking (2PL) [20]. Every item in a transaction obtains the same logical timestamp. This 2PC protocol is layered on top of TAO, and write transactions are opt-in.

Since items on different shards are independently and asynchronously replicated to TAO cache replicas, a concurrent read transaction can observe partial updates before all parts of a write transaction are fully replicated. This is especially true if a failure occurs during 2PC and recovery takes place. While the vast majority of TAO replicas (99.99%) are less than 60 seconds stale [45] and

the failure recovery process typically finishes within a few seconds (see Figure 7b), billions of reads can occur within this time frame.

Bidirectional associations. Bidirectional associations fall in a special category of failure-atomic transactions. These data structures are pairwise edges in the social graph (e.g., COMPOSED and COMPOSED_BY in Figure 2). Since edges in a pair are sharded by their source node, they can be stored on different shards. They are ubiquitous in TAO, consisting of 52% of association types and 56% of the association write volume. TAO originally did not guarantee atomicity of bidirectional association updates (Section 4.2 of [21]). However, developers expect the forward and inverse edges to have the same data and timestamp. Consequently, TAO gradually strengthened the guarantees of bidirectional associations and started to consider them a special category of (potentially cross-shard) transactions. Most of these transactions can complete in one round (i.e., without the first phase of 2PC) because they have no preconditions, such as uniqueness or existence. A background fixer process now ensures failure atomicity. For the rare but important cases in which these edges require some invariant to hold, developers use the cross-shard transactions previously described.

We highlight bidirectional associations because of their prevalence. Any mechanism we implement for atomic visibility needs to work efficiently for bidirectional associations. While applications typically read a pair of edges from only one side, more complex queries may indirectly involve both edges.

2.3 Design goals

In this paper, we develop read transactions that allow users to perform batch reads over TAO with atomic visibility guarantees. In the design of these protocols, we have several important considerations, given the large-scale, read-optimized environment at Facebook.

First, we need to ensure that most queries can still be served with low latency. Ideally, only applications that want these semantics and only their transactional operations incur any associated performance and / or storage costs.

Second, we should allow applications to opt into transactional features and stronger guarantees late in their development cycle. This retroactive optimization is a recurring pattern we have seen at Facebook, and we wish to enable users to switch isolation levels without changing their application code.

Third, our implementation should ideally be extensible to additional data stores without core replication protocol changes. In theory, this will allow us to apply these ideas to other data storage systems as needed.

Finally, we want transactional features to be suitable for gradual rollout. Changes to our mature production system often take several years, and we release them to a handful of applications at a time.

These goals motivate the development of a layered approach to our protocol design (Figure 1). Specifically, we can decouple the concerns of availability and replication (provided by TAO) from atomic visibility guarantees (provided by our protocols), similar in spirit to the "bolt-on" approach [18]. Our protocols are more complex than the original proposal: we extend the TAO caching and database layers to extract desired guarantees and optimize for performance, instead of solely using the data stores's read / write APIs. Nevertheless, this separation is practical for large-scale systems and simplifies reasoning about consistency and isolation.

Algorithm 1: Atomic visibility checker

Input: read set $\mathcal{R}$

```
read ⟨timestamp ts, metadata md⟩ for each r ∈ R

// Get the latest timestamp of each data item from md
ts_latest ← {}
for r ∈ R do
    for k_txn ∈ r.md do
        ts_latest ← max(ts_latest[k_txn], r.ts)

// Check that each read value has the latest timestamp
for r ∈ R do
    if ts_latest[r.key] > r.ts then
        return false
return true
```

3 MEASUREMENT STUDY

To assess the impact of a layered solution for atomic visibility, we perform a measurement study of this guarantee at scale. Specifically, we measure the frequency of atomic visibility violations of naïvely batched reads and analyze the causes of these anomalies. Since we do not yet have read transactions in production, our study is based on the distribution of multi-key, production requests. Our findings demonstrate the need for atomic visibility in a read transaction API.

3.1 Measuring atomic visibility violations

To measure potential atomic visibility violations, we first profile multi-key, production read and write traffic. Based on this profile, we mirror write transactions to a test tier, generate batched reads, and invoke our atomic visibility checker to record any anomalies. We also track the duration of these anomalies before full replication and recovery logic occurs.

Modeling production data. We extract probability distributions from Ent to generate load distributions for our measurement study. Currently, *Ent queries* that involve multiple sub-queries and accesses to TAO represent the read transactional boundary we consider supporting in the long term, so we use them to model read transactions. Similarly, *Ent changesets* capture transactional intent for writes. We focus on the main characteristics of changesets and queries: the number of operations, the number of data items involved, the number of distinct shards, and the success rate.

Generating workloads. We use a load generator to send write transactions to test shards, which have the same TAO and underlying database setup as in production within a single datacenter. We inject a small percentage of failures, in accordance with the probabilities gathered from production. The generator sends batched reads as "read sets" to our atomic visibility checker.

We generate two types of read workloads: one based on the query pattern profile mentioned above to evaluate production scenarios and the other with read sets exactly overlapping transaction write sets so that we measure the worst case scenario for atomic visibility violations (Figure 7).

To extract probability distributions representative of TAO's workloads, we study Ent datasets gathered over 30 days. We only consider Ent queries (reads) spanning more than one data item to represent read transactions. Similarly, we filter for changesets (writes) involving two or more data items since these operations could be

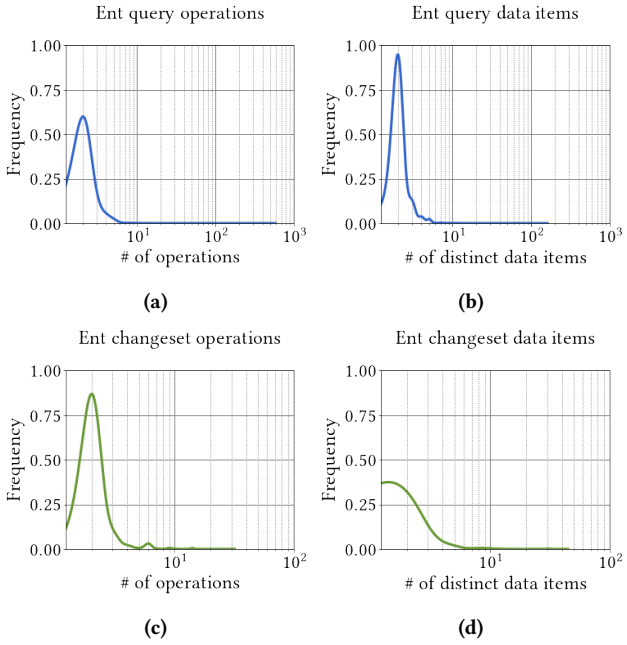


Figure 5: (a), (b): Ent queries typically span fewer than 10 operations and data items, but some queries can touch many keys. (c), (d): Most Ent changesets involve a few operations and data items.

completed as transactions. Out of the filtered data, we find that most Ent queries and changesets are limited to several data items (Figure 5). On average, Ent queries contain 2.4 operations and 2.2 data items while changesets contain 2.4 operations and 2.1 data items. However, the Ent query distributions have long tails, containing up to 587 operations and 162 data items. This wide range demonstrates the complexity of Facebook's application logic. Both Ent queries and changesets involve data items on 1 to 10 database shards. Ent changesets have a failure rate of 2.22%, which we use as the manually injected failure rate. Note that this rate captures issues such as client-side timeouts, malformed configurations, and precondition failures (e.g., deleting an already-deleted object), so they represent an upper bound on the actual failure rate.

Atomic visibility checker. Since TAO's write transaction protocol is similar to RAMP, in which all items in a write transaction are assigned a unique logical timestamp that is monotonically increasing for each item, we use an algorithm similar to the RAMP-Fast protocol [17] to detect fractured reads. Unlike the original protocol, we must fetch data items and transactional metadata separately due to how information is currently stored in TAO and the underlying database. If some information is not available (due to replication lag), we retry these reads. This checker is responsible for verifying whether naïvely reading a set of data items violates atomic visibility.

As shown in Algorithm 1, the checker first reads the items (individually but in parallel) along with their timestamps and transactional metadata, if they exist. Specifically, the checker attempts to fetch the most recent transactional write set for each data item. The information allows us to detect any atomic visibility violations.

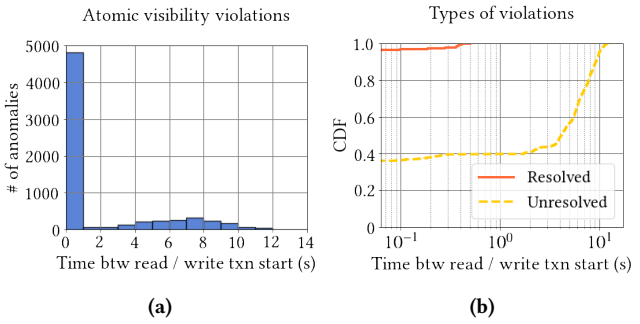


Figure 6: With generated production read sets, (a) anomalies can occur for read transactions up to 13 seconds after a write transaction starts. (b) Anomalies that can be resolved (orange) are ones that overlap with successful write transactions and occur within 500ms of the write transaction start.

As an example, suppose transaction T_1 writes to items x and y with timestamp 1 and transaction T_2 writes to items x and y with timestamp 2. A read set $\mathcal{R} = \{x, y\}$ triggers the checker. In the initial reads, the checker obtains x with timestamp 1 and y with timestamp 2. It also fetches the write sets of the transactions for those versions ($\{x_1, y_1\}$ for x_1 and $\{x_2, y_2\}$ for y_2). The checker now observes that the results x_1 and y_2 include only part of T_2 (namely y_2) and exclude x_2. Thus, these read results violate atomic visibility.

When the checker finds an anomaly, it continues re-reading the input read set and re-checking the results, in order to gauge how long it takes for write transactions to be completed (possibly with failure recovery) and fully replicated.

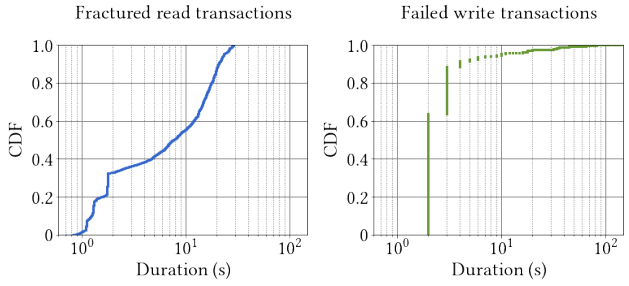
Bidirectional associations. We evaluate potential atomic visibility anomalies for bidirectional associations separately because most updates to these edges are a special case of write transactions that do not need 2PC. We measure the frequency and duration of of bidirectional associations failures. This data allows us to quantify the window in which read anomalies are possible.

3.2 Analysis

Atomic visibility checking results. Out of the 10.1M read batches and 1.5M write transactions we generate, we find 6,433 of our reads are fractured: almost 1 in every 1,500 transactions violates atomic visibility. For these anomalies, we measure the time between when each write transaction starts and when the subsequent read transaction begins. Fractured read transactions occur up to 13 seconds after the write transactions they overlap with, as shown in Figure 6a.

We find that 45% of these fractured reads last for only a short period of time (i.e., naïvely retrying within a few seconds resolves these anomalies). After a closer look, these short-lasting anomalies occur when read and write transactions begin within 500 ms of each other. For these atomic visibility violations, their corresponding write transactions were all successful. In other words, these were transient anomalies due to replication delays.

The other 55% of these atomic visibility violations could not be fixed within a short retry window and last up to 13 seconds. For this set of anomalies, their overlapping write transactions needed to undergo the 2PC failure recovery process, during which read anomalies persisted.



(a) (b)

Figure 7: With transactional write sets as exact read sets (worst case scenario), (a) fractured reads do not persist for longer than 30 seconds in our experiments, because (b) most write transactions are fixed within this period. Note that the recorded timestamps have second granularity.

Worst case scenario results. As previously mentioned, we also check read sets that overlap exactly with transactional write sets to gauge how long atomic visibility violations last in the worst case. We find that read anomalies in our experiments do not persist past the 30 seconds mark (Figure 7a). This correlates with the time it takes for the 2PC failure recovery process to complete in most cases (Figure 7b), indicating that anomalies are resolved when write transactions eventually settle.

Bidirectional association results. Up to 2.0% of bidirectional associations need to be fixed after failure. In hot spot scenarios, bidirectional associations may take minutes to fully settle. While reads to both edges are currently unlikely, there is a substantial window of time in which atomic visibility violations can be returned, demonstrating the importance of providing stronger read guarantees for bidirectional associations.

3.3 Discussion

Our measurement study shows that atomic visibility violations can occur at a rate of 0.06% for naïvely batched reads. Seemingly small, this infrequent rate can be a significant source of complexity for our product developers. These read anomalies are rare and nondeterministic, making them difficult to detect, reproduce, and investigate. Atomic visibility is integral to an intuitive read transaction API that will greatly simplify the development process for engineers.

Write transactions that need to undergo 2PC failure recovery (0.03% of write transactions) increase the scope and duration of atomic visibility violations. Causes of failures range from client-side contention or precondition failures to infrastructure hiccups. These cases result in a majority (55%) of the anomalies we find, which can persist for up to 30 seconds. This is a strong motivator for having multiversioning even if limited to only transactional writes. For example, if we know a write transaction has yet to recover, we can return the prior version of an item on a read.

Any decrease in write availability (e.g., from service deployment, data center maintenance, to outages) increases the probability that write transactions will stall, leading in turn to more read anomalies. This probability also grows with the size of a transaction's write set, as more shards are involved. It is unrealistic to completely remove write unavailability and the ensuing atomic visibility violations if we only naïvely batch reads.

Though write transaction failures are rarer in production than in our measurement study (since we consider the worst case scenarios), atomic visibility violations introduce considerable complexity for product developers. Therefore, we need to pursue atomic visibility in a read transaction API.

4 READ ATOMIC ISOLATION FOR TAO

In this section, we motivate our choice of Read Atomic (RA) isolation for TAO over other isolation guarantees. We then provide an overview of RAMP transactions which provide RA isolation and highlight the challenges of adopting them at Facebook. We present a novel variation of the RAMP protocols, RAMP-TAO, that ensures atomic visibility in a read-optimized, geo-distributed setting. RAMP-TAO can quickly determine if a data item is a part of a recently completed write transaction, enabling most reads to complete in one round trip in the fast path and with minimal metadata overhead.

4.1 Read isolation

When deciding on an isolation model for read transactions in TAO, we must account for the fact that reads dominate TAO's workload and must therefore be especially efficient. In contrast to TAO's write transactions, we want read-only transactions to be non-blocking. Supporting stronger isolation models such as Snapshot isolation (SI) in our read-optimized environment—with low latency (most requests served locally) and cacheability for hot spot tolerance [14]—remains challenging. Reading under SI requires systems to confirm the presence or absence of certain updates, resulting in large dependency sets even if the representation is compact (e.g., a timestamp). Moreover, SI reads must hide newer updates, such as by using multiversioning [24, 50], and increase the likelihood that data needs to be fetched cross-region in order to satisfy versioning constraints.

Among weaker isolation levels, Read Committed (RC), which prevents access to uncommitted or intermediate versions of data items, is often chosen as the default isolation level of many data stores [16]. However, RC allows fractured reads of partial updates, so it does not guarantee atomic visibility.

In contrast, Read Atomic isolation, which sits between RC and SI in terms of strength, provides atomic visibility while maintaining scalability. Moreover, the dependency set required for RA isolation is bounded to the write sets of data items in the same write transaction(s). This isolation model works well within TAO's constraints.

While RA isolation is a practical solution for TAO and is the subject of our study in this paper, it is not sufficiently strong for all use cases. For example, point-in-time snapshots would be useful for analytical queries, but this functionality is not supported by RA. This isolation model also does not prevent read-write conflicts. However, we have found RA isolation to be a practical, incremental solution in the service of strengthening isolation guarantees on TAO. As described in Section 2.1, atomic visibility is sufficient for many of our potential use cases.

4.2 Design challenges

To implement Read Atomic isolation, we consider the RAMP protocols [17], which are the only algorithms we are aware of that

specifically guarantee this isolation level for databases. RAMP offers availability, scalability, and efficient performance while providing atomic visibility. Although the original RAMP paper does not consider non-transactional workloads, the impact on single-item requests is minimal because RAMP reads are non-blocking. To prevent reads from stalling, RAMP readers rely on multiversioning support to ensure that requested versions are always available.

The read protocol consists of two phases. In the first round, RAMP sends out read requests for all data items and detects non-atomic reads. In the second round, the algorithm explicitly repairs these reads by fetching any missing versions. RAMP writers use a modified two-phase commit protocol that requires metadata to be attached to each update, similar to the mechanism used by cross-shard write transactions on TAO.

While promising, the RAMP protocols cannot be directly applied to TAO. There are three main issues: performance requirements for read transactions, metadata storage constraints, and the lack of multiversioning. We discuss each of these challenges in turn below.

Mostly fast read transactions. To serve TAO's challenging workloads at scale, most read transactions should have comparable latency and storage overhead to existing read queries. In particular, we desire low latency for read transactions that access non-transactional data, since few operations are involved in transactions (currently less than 3% of writes on average). Reading items that were not updated by a write transaction is atomically visible by definition, so we do not need to fetch metadata and check for read anomalies on these results. In contrast, the original RAMP protocols consider purely transactional workloads and require that all read transactions verify that their results are atomically visible.

Furthermore, most read transactions should finish in one round of communication with the cache to enable performance in line with current TAO requests. As a result, we prefer the variation of RAMP that enables one round reads by fetching the complete transactional write set of each data item (i.e., RAMP-Fast) [17].

Handling hot spots is one of the major challenges of a social networking workload [14]. As a result, our solution should ensure cacheability of intermediate or final read results so that we mostly serve data from the local region. This challenge is not covered in the original RAMP paper, but one we must address in our setting.

Accessing transaction metadata. Obtaining the complete set of transaction metadata for each data item is challenging in TAO because of the size of this metadata and its access latency. The original RAMP protocols do not consider the costs of replication for availability [17]. However, in our environment, maintaining multiple copies of metadata, in both cache and database replicas, would generate heavy storage overheads. Moreover, updating this metadata across different replicas would incur latency penalties.

For cross-shard write transactions, we weighed several design choices for metadata content and placement. In particular, we wanted to minimize the overhead of storing metadata for the large majority of applications that did not use transactions. The cost of storing transaction metadata in the cache is relatively high compared to the underlying database, so we opted to store metadata in the database to maximize cache capacity for serving requests. We also decided to store metadata "out-of-line" in a separate table within the database instead of inline with each row to avoid write amplification. This table holds only information on participating

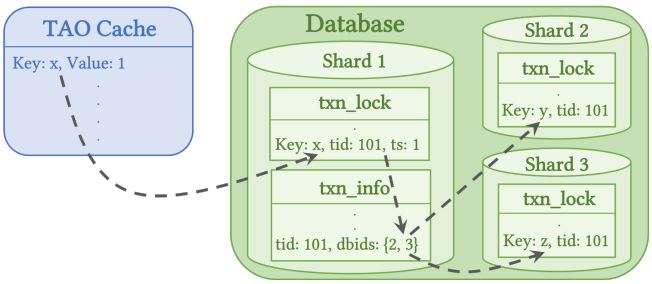


Figure 8: Transaction metadata layout. The arrows show the request path to obtain the most recent transactional write set for a key x. We first fetch the transaction id to identify the participant shards and then acquire the various keys involved in the transaction. In this case, the write set for key x is $\langle x, y, z \rangle$.

shards rather than specific data items to further reduce storage overhead. Figure 8 depicts our transaction metadata placement.

While these choices save space and improve cache performance, they complicate the problem of piecing together the write set of a transaction for use in atomic visibility checking. The current schema requires multiple round-trips to the database to gather all the necessary metadata. Instead, we seek a mechanism to access this information from within the cache for most cases to serve read transactions with low latency. When we do need to communicate with the database, we should be able to amortize these requests.

Lack of multiversioning. Finally, the absence of multiversioning support on TAO makes it difficult to support RAMP transactions. The RAMP protocols rely on access to multiple versions from the underlying database to guarantee termination of read transactions in at most two rounds of communication. Accordingly, a direct implementation of RAMP would allow us to detect atomic visibility violations but does not enable us to fix all of them. Since TAO offers Ticket-inclusive reads [45]—at-or-after reads based on version, which provide a lower bound on writes that should be visible—we can naïvely retry reads until we have an atomically visible result. However, we cannot guarantee that a read transaction will terminate with this strategy.

As an example, consider a situation in which a cross-shard transaction T_1 writes to both x and y. Simultaneously, T_r reads these two keys while T_1 is writing. Specifically, T_r reads from x after T_1's write to x has committed but reads from y before T_1's write to y has committed. Thus, T_r's first round reads return $x = x_1$ and $y = \perp$, a RA violation. Consequently, T_r attempts a second round of reads, but in the meantime, T_2 writes to both x and y under two-phase commit. If T_r reads from x and y while T_2 is writing, it could read $\{x = x_1, y = y_2\}$ or $\{x = x_2, y = y_1\}$ on its second round, both of which violate RA. In this case, T_r can retry the read again, but the requested version will not necessarily be present.

Since only a single version is available for each data item, the version needed to satisfy atomic visibility can be continuously overwritten, so the number of communication rounds can be unbounded. Therefore, we cannot guarantee read transaction termination without multiversioning. However, storing multiple versions of all data items in TAO imposes significant storage overhead and additional

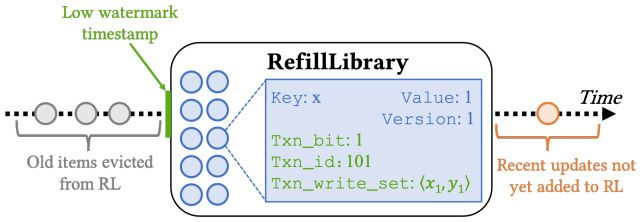


Figure 9: By storing transactional information in the RefillLibrary, we enable most read transactions to complete in one round. We add the fields in green for RAMP-TAO.

complexities in the optimized read path and garbage collection of old versions, which would affect users that do not need stronger guarantees. We present a practical solution to this problem in the next section and demonstrate it incurs minimal overhead in Section 5.2.

4.3 Protocol

To address these challenges, we present the RAMP-TAO protocol, which provides atomic visibility with negligible impact on TAO's performance and storage overheads. At a high level, we leverage the fact that naïvely reading from TAO is atomically visible by default once all updates in a transaction are globally replicated. Thus, RAMP-TAO only needs to check for anomalies on recently modified transactional data. To identify these items, we extend the functionality of the RefillLibrary, a recent writes buffer in TAO. We further modify this buffer to store transaction metadata and multiple versions to ensure read transactions can terminate efficiently. As a result, RAMP-TAO requires minimal changes to TAO, and its ability to detect atomic visibility violations is also useful for monitoring purposes. We now discuss this protocol in depth and illustrate how it solves the three aforementioned challenges.

Atomic visibility check. Since data items not written to by the same transaction are read atomic by definition and a large portion of data returned by TAO is already atomically visible (Section 3), the first part of our protocol performs a naïve batched read (Algorithm 2, lines 3–5). RAMP-TAO then locally checks whether these results already satisfy atomic visibility (lines 7–21). To do this, we need information from our system that indicates whether an item has been modified by a write transaction. We use the `RefillLibrary` (Figure 9) to store this information.

The RefillLibrary is a metadata buffer recording recent writes within TAO, and it stores approximately 3 minutes of writes from all regions. The RefillLibrary and the main cache are updated by the same at-least-once replication stream from Wormhole [44]. The RefillLibrary was originally introduced to maintain cache consistency, and it is currently only accessed during cache misses. In the case of a cache miss, the main cache (on the same server) must actively fetch newer updates from upstream and also query the RefillLibrary for information. As a result, the main cache can contain items more recent than what is in the buffer. In practice, the RefillLibrary is almost always up-to-date with the cache.

To provide support for limited multiversioning in RAMP-TAO, we extend the RefillLibrary to include a bit for each write to indicate whether it was written as part of a transaction. For each data item in a read transaction, we attempt to fetch this transactional bit and

Algorithm 2: RAMP-TAO

```
1  procedure READ_TXN(I : set of items):
2      ret, ret_txn, ts_latest ← {}
3      // Read all values from TAO
4      parallel-for i ∈ I do
5          ret[i] ← TAO_READ(i)
6
7      // Check if any items were updated by transactions
8      if {i ∈ ret : i.updated_by_txn} == {} then
9          return ret
10
11     // Detect and fetch any missing items
12     while !timeout do
13         // Find the latest timestamps from txn metadata
14         for r ∈ {i ∈ ret : i.updated_by_txn} do
15             for k_txn ∈ r.md do
16                 ts_latest[k_txn] ← max(ts_latest[k_txn], r.ts)
17
18         needed_ts ← {}
19         for i ∈ I do
20             if ts_latest[i] > ret[i].ts then
21                 needed_ts[i] ← ts_latest[i]
22
23         newer_ts ← {} // Items newer than requested
24         parallel-for i ∈ needed_ts do
25             // At-or-after read
26             ret[i] ← TAO_READ(i, needed_ts[i])
27             if ret[i].ts ≠ needed_ts[i] then
28                 newer_ts[i] ← ret[i].ts
29         if newer_ts == {} then
30             return ret
31
32     return ERR_TIMEOUT
```

any other transaction metadata from the RefillLibrary (lines 3–5). If a data item is in the buffer, examining this bit allows RAMP-TAO to determine if any transactional data items are being read (lines 7–9). If a data item is not in the buffer, there are two possibilities: either it has been evicted (aged out) or it was updated too recently and has not been replicated to the local cache. To distinguish between these two cases, we fetch and compare the logical timestamp of the oldest data item (the low watermark timestamp) in the RefillLibrary with that of the data items being read. If the items are older than the low watermark timestamp (Figure 9), they must have been evicted from the buffer, and the writes that updated these items have already been globally replicated. Otherwise, our local RefillLibrary does not have the most recent information and must fetch it from the database. From the example in Section 4.2, if T_r reads $\{x = x_2, y = y_2\}$ from the main cache, but these data items are no longer in RefillLibrary, RAMP-TAO can compare timestamp 2 with the low watermark timestamp (let us say 5 in this case) to determine that these items have been evicted from the buffer. Thus, we can safely return these results since T_2 must have been fully replicated, and thus, the results are atomically visible.

Gathering transaction metadata. For read transactions that contain recently updated transactional data items, RAMP-TAO uses the full write set for each item to determine atomic visibility (lines 13–21). We modify the RefillLibrary to store transaction metadata by adding fields for the transaction id, timestamp, and participating keys. By making this information cacheable, we facilitate hot spot

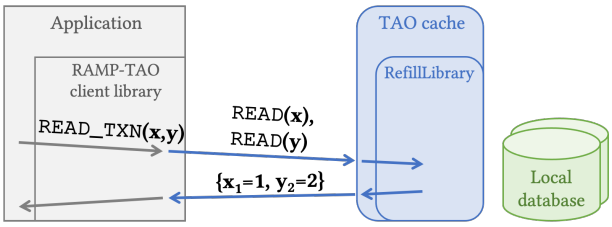


Figure 10: Example execution of the fast path. A read transaction of $\langle x, y \rangle$ is sent to RAMP-TAO. The first round read returns results indicating the two keys that have not be recently updated to by a write transaction. The result is atomically visible by default, and RAMP-TAO can return after one round.

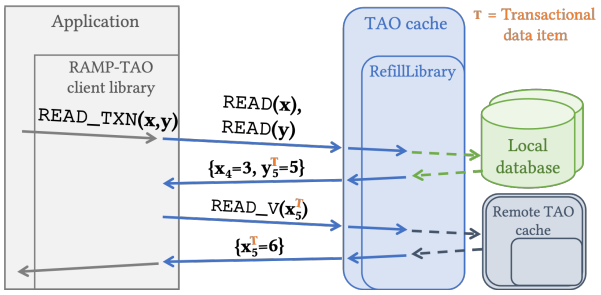


Figure 11: Example execution of the slow path. A read transaction of $\langle x, y \rangle$ is sent to RAMP-TAO. The first round results represent an atomic visibility violation because x and y were updated together in a write transaction. RAMP-TAO determines x_5 is needed and fetches this with a second round of communication before returning a read atomic result.

tolerance for read transactions and avoid several round trips to the database. This metadata is fetched in the first round read alongside the transactional bit, and RAMP-TAO uses the write sets to detect atomic visibility violations. If this information is not available in the RefillLibrary, we obtain it from the database. However, this extra query is rarely needed because the buffer is almost always up-to-date with the cache. Furthermore, we can amortize the cost of these metadata reads across subsequent TAO queries.

Multiversioning in cache. When RAMP-TAO detects a fractured read, it sends requests to TAO for versions that would ensure an atomically visible result. We observe that we only need multiversioning for data items *recently* updated *by transactions*. Data items not written to by transactions trivially satisfy atomic visibility. Older transactions that have already been fully replicated also do not violate atomic visibility, since all parts of those transactions can be observed.

While TAO does not provide full multiversioning, we extend the RefillLibrary to store multiple versions of recent, transactionally-updated data items. The memory impact of this change should be small given that the RefillLibrary already stores 3 minutes of recent writes, and we verify this claim in Section 5. Since the retention of the RefillLibrary is higher than the failure recovery time of write transactions (at most 146 seconds including replication lag, Section 3), any transactional updates that are evicted and garbage collected will have been fully replicated. Adding bounded multiversioning to the RefillLibrary allows RAMP-TAO to request specific versions needed to satisfy atomic visibility.

As discussed in Section 4.2, the primary challenge from the lack of multiversioning is read termination. The RefillLibrary's limited retention of multiple versions decreases the chance of non-termination. However, RAMP-TAO reads of the requested versions (line 26) may be hindered by in-progress write transactions or asynchronous replication. In the Section 4.2 example, T_r should be able to read $\{x = x_2, y = y_2\}$ from the RefillLibrary if T_2 has completed and globally replicated. However, if T_2 is still in progress or one of the keys has not yet been replicated, T_r cannot return an atomically visible result.

Thus, we present an optimization of our protocol. Instead of having TAO always return the newest available versions (line 26), we can ask the system to return slightly older ones to satisfy atomic visibility (note that we omit the details of this optimization for clarity in Algorithm 2). The main advantage of trying to obtain less up-to-date versions is that the RefillLibrary will almost certainly have these items. In the example from Section 4.2, RAMP-TAO could return $\{x = x_1, y = y_1\}$ instead of waiting for T_2 to complete and replicate.

This strategy enables RAMP-TAO to terminate as long as the older versions are still in the RefillLibrary after the first round read. Since the buffer's retention window is much longer than the time it takes for a round trip to TAO (typically well under one second [21]), an item that has been read on the first round is unlikely to have been evicted before another round occurs. Since applications already tolerate stale read results under TAO's eventual consistency, returning an older, atomically visible result is preferable to the possible increase in latency and number of rounds while trying fetch the most recent versions.

With this optimization, read termination is almost always guaranteed. For the extreme cases in which the RefillLibrary does not have the necessary versions (e.g., the requested version is not yet replicated but previous versions have been evicted), RAMP-TAO falls back to reading from the single-versioned database to obtain the latest version. If the versions from the database still does not satisfy atomic visibility, we either naïvely retry the reads with a backoff and timeout or hold locks on contended keys during reads to guarantee termination, although we have not found the need to do so. In these exceedingly rare cases, we opt to provide a practical solution for termination: the transaction fails and it is up to the client to retry. However, we never return anomalies to the client. In the future, we will explore using client-exposed multiversioning support in the database to always guarantee termination.

End to end protocol description. Putting the pieces together, we describe the request flow of our RAMP-TAO protocol for when requests can be served locally (fast path) and when communication to the database or a remote region is required (slow path). Read transactions that satisfy atomic visibility by default can complete after one round of communication with the local cache under RAMP-TAO (Figure 10). If the RefillLibrary indicates that a read transaction contains no data written by a write transaction within the last 3 minutes, we know the result is read atomic by definition and can return to the client immediately. Otherwise, we check our

results using transaction metadata from the RefillLibrary and can return if the read is atomically visible.

Requests can take the slow path when information is not available in the local cache or a fractured read is detected, as shown in Figure 11. When the RefillLibrary does not yet contain the most recent transactional updates, RAMP-TAO requires a query to the database to fetch the necessary metadata. If an atomic visibility violation is found, we require another round of communication to TAO. If the needed versions are unavailable in the local cache, we must fetch them from the database or another region. These additional queries are only required if the local RefillLibrary is not up-to-date. With our optimization, we can fall back to reading older versions in the RefillLibrary that satisfy atomic visibility. Therefore, the second (dashed, grey) query out of the RefillLibrary in Figure 11 will rarely be needed. We evaluate the frequency and latency of these requests in the next section.

4.4 Optimization for bidirectional associations

Given the ubiquity of bidirectional associations in our data model, we need to ensure our protocol is especially efficient for these pairs of edges at scale. As previously mentioned, bidirectional association writes are considered a special category of write transactions (Section 2.2), since they always involve two specific keys (the forward and the inverse edges). Most of these writes do not have preconditions like uniqueness, which means they do not need the first phase of 2PC—the committed edge of either side serves as a persisted record of that transaction. This obviates our need for storing transactional metadata for most writes to these paired edges.

On the read side, we optimize for the case when a read transaction includes a pair of bidirectional associations. Since both sides contain the same data and metadata (logical timestamp), we can use the results of one edge to repair the other. When an anomaly is detected, RAMP-TAO simply replaces the data in the older side with the information in its more up-to-date inverse. This additional processing is purely local. Thus, under the RAMP-TAO protocol, bidirectional association reads always complete in one round.

5 EVALUATION

We proceed to experimentally demonstrate the feasibility of RAMP-TAO for the Facebook production environment. We implement a prototype of our protocol and find that over 99.93% of read transactions can be served in one round. We also benchmark the performance and memory overheads of our proposed changes to TAO and show that their effect is minimal. These results validate that we can provide atomic visibility efficiently for our system.

5.1 RAMP-TAO prototype

Our RAMP-TAO prototype is implemented in C++ and uses Thrift [3], a serialization and RPC framework. We make a few adjustments from the protocol description in Section 4 due to what is currently available in our production environment. We query the local-region database for transaction metadata, since it is not yet available in the TAO cache. Also, our prototype does not have access to multiple versions of data items, so we rely on the timeout in Algorithm 2 to retry reads for up to 10 seconds, as most write transactions complete within this period (Section 3).

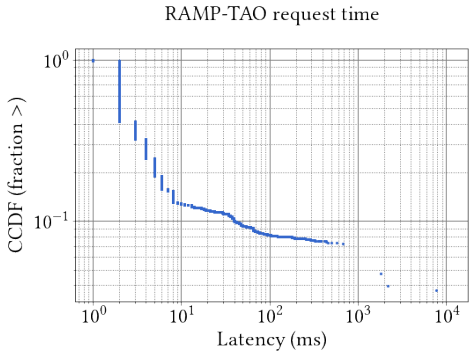


Figure 12: Most read transactions finish in under 100ms with our RAMP-TAO prototype. These latencies will decrease as we adapt the RefillLibrary for our protocol.

5.2 Results

We evaluate the performance of our prototype and demonstrate that RAMP-TAO will allow the majority of read transactions to complete with latency comparable to current TAO reads when rolled out in production. The memory overhead of our protocol also does not significantly impact our system. We generate workloads for read transactions using a similar mechanism as described in Section 3.

Performance overhead. Our prototype serves over 99.93% of read transactions in one round of communication. Even when a subsequent round is necessary, the performance impact is small and bounded to under 114ms in the 99^{th} percentile (Figure 12). Our tail latency is within the range of TAO's P99 read latency of 105ms for a similar workload. We note that these are the worst-case results for RAMP-TAO because the prototype currently requires multiple round trips to the database for transaction metadata. Once the changes to the RefillLibrary are in place, the large majority of the read transactions can be directly served with data in this buffer and will take no longer than a typical TAO read.

For the 0.06% of read transactions that require more than one round, the RefillLibrary will be able to serve most of these requests once it supports multiversioning. In the rare instances in which the necessary data is not available in the RefillLibrary, we query the database for information, similar to our current prototype. The frequency of these cases should be small, considering the freshness of data in the RefillLibrary and our protocol optimization (Section 4.3). Given that our current results include the latency of multiple database accesses (Figure 12), most read transactions can still complete quickly even if some information is not in the RefillLibrary.

Finally, database unavailability is an issue that the RefillLibrary cannot address. While read transactions to these keys may stall during this period, these types of failures are very infrequent and will have a limited impact on overall read latency. Given these occurrences are rare, we opt to allow the application to decide what to do after a timeout rather than returning fractured reads.

Memory overhead. RAMP-TAO requires three pieces of information to be added to the RefillLibrary: a bit indicating if a data item was transactionally updated, write transaction metadata, and multiple versions. While these modifications decrease the buffer capacity, our benchmarks show that the memory overhead of our protocol is minimal.

The transactional bit allows the RefillLibrary to quickly detect if our first round read result already satisfies atomic visibility. This change actually has no impact on the capacity of this buffer: data items are allocated based on size class, so packing an additional bit will not cause them to be re-classified.

To access transaction metadata faster and to avoid database accesses, we need to add this information to the RefillLibrary. For each transactional item, we include the transaction id (16 bytes) and the transaction write set (each key is no more than 24 bytes, and most write sets contain fewer than 10 keys). Compared with a direct approach (such as in the original RAMP protocols) in which transaction metadata is stored and propagated to all replicas, our strategy has several advantages. We only incur memory overhead in cache for *transactional* data items (3% of writes) in the *recent past*. Additionally, since we can always fall back to the slow path, the RefillLibrary has the flexibility to make the tradeoff between extra latency on reads and buffer space. For example, if a data item is updated in a transaction with an unusually large write set, the RefillLibrary has the option to store only the transactional bit without the rest of the metadata. Overall, this overhead is negligible.

The final source of memory overhead is storing multiple versions of a transactional data item in the RefillLibrary. Under current workloads, 3% of writes are transactional, and at most 14% of additional versions need to be stored for these updates within the RefillLibrary's retention window. Thus, keeping multiple versions in the buffer would lead to a 0.42% overhead with existing traffic. In other words, even in the worst case, the RefillLibrary only needs to store less than one additional version, on average, per transactional data item. We note that this overhead can vary widely with the workload and it is the extreme case that matters.

5.3 Discussion

Our prototype and benchmarks validate the feasibility of using RAMP-TAO in production. We show that the performance and memory overheads of this protocol are inconsequential in practice. Importantly, non-transactional requests to TAO are virtually unaffected since RAMP read transactions are non-blocking. Furthermore, the RefillLibrary will not become a bottleneck for reads since only cache misses from non-transactional requests currently access it (Section 4.3), and transaction traffic will be relatively small.

Our solution minimizes the number of requests that must go cross-region or to the database so that over 99.93% of read transactions can take the fast path. Even for reads on the slow path, the additional latency incurred is comparatively low and in line with current TAO requests that must go upstream.

RAMP-TAO requires minimal changes to the existing infrastructure and is suitable for incremental rollout. The cross-shard write transaction protocol on TAO is very similar to the RAMP two-phase write protocol, and all the necessary metadata is already stored. We leverage the existing RefillLibrary to provide the required information for detecting and repairing atomic visibility violations. Using this buffer for limited multiversioning is an efficient strategy since we only need to access specific versions before replication completes. Once a write transaction is fully replicated, reads of those updates automatically satisfy atomic visibility (with respect to that transaction).

As a side benefit, the first half of the RAMP-TAO protocol is conducive to monitoring atomic visibility anomalies and serves as an effective verification mechanism as well as detection tool for new use cases.

Furthermore, our approach with RAMP-TAO is generalizable: by storing a bit within the cache, we can quickly identify whether a read transaction can go down the fast path. This optimization is especially important for read-optimized, geo-replicated systems for which cross-region requests are expensive. Most of these systems implement a caching layer for performance because accessing persistent storage is slow [11, 21, 38]. Our protocol maximizes the proportion of requests served by the local cache to avoid the high latency of going cross-region or to the database.

6 LESSONS LEARNED

Stronger guarantees only when necessary. Given TAO's demanding workloads, any degradation in service availability, request latency, or resource efficiency to support stronger guarantees should be borne solely by the callers opting in to higher isolation levels. Similar to the case with stronger consistency [45], only a small but important fraction of requests benefit from stronger transactional semantics. For these applications, we offer the option of higher isolation levels and allow developers to choose whether to incur the resulting performance costs.

By layering RAMP-TAO on top of the underlying data stores, we can isolate the impact of stronger guarantees and take advantage of the performance benefits that TAO already provides. Our layering approach also limits changes to TAO's mature infrastructure and enables isolation guarantees to be strengthened with relative ease. The architectural separation proposed in [18] can be an effective strategy for large-scale systems.

Working with read-optimized systems. Providing stronger isolation and consistency guarantees for read-optimized systems boils down to a staleness check of local data and infrequent cross-region requests to fetch any missing pieces. This recurring design pattern is a natural consequence of a read-heavy, geo-replicated environment and can be found in multiple systems within Facebook such as FlightTracker [45]. The staleness check must: (1) rely on local data to avoid cross-region communication in most cases, (2) result in few false positives so that we minimize any extra work, and (3) conducive to incremental repair so that stronger guarantees can be provided. Our RAMP-TAO protocol also follows this pattern.

RAMP-TAO completes a local check for atomic visibility in its first round by relying on the RefillLibrary and only goes cross-region to obtain any missing versions that have not yet been replicated. The RefillLibrary's garbage collection (low watermark) timestamp is sufficient to avoid most false positives and distinguishes between data items that have been evicted and those that have not yet been added. Furthermore, the RAMP protocols use a strategy of detecting then repairing read anomalies, which enables incremental repair of stale results.

Read and write availability. Our results in Section 3 show that write (un)availability can impact read availability. The interaction of different systems can further complicate the implications of write availability. With the emergence of FlightTracker [45], we now encode recent write metadata in Tickets and attached them to

TAO reads to provide RYW consistency. The Ticket-inclusive (i.e., at-or-after) reads advance timestamps and versions more rapidly. This behavior amplifies the effects of write unavailability to reads in the transactional context. If a cross-shard write transaction stalls or has not yet undergone failure recovery, the versions requested by Tickets may not be available.

To address this issue in the short term, we could preempt (roll-back) this write transaction. If FlightTracker requests a version that is unavailable, the system can fallback to aborting the relevant transaction to prevent lack of termination. RAMP-TAO can also use this strategy to complete reads in the case of severe write unavailability. When full multiversioning is supported, we can allow reads to always proceed without stalling.

7 RELATED WORK

Transactional semantics. There has been a pattern of retroactive addition of transactional semantics to existing data stores. For example, Cassandra 2.0 [1] has added failure-atomic transactions. DynamoDB [5] has also recently offered ACID transactions that use a centralized transaction coordinator. MongoDB [4] has included transactions that support Snapshot Isolation. In line with this trend, TAO added failure-atomic write transactions, and this paper details how atomically visible read transactions can be layered on top.

Many recent data stores provide strong consistency and isolation guarantees as well as transactional APIs. CockroachDB [50], etcd [8], and Yugabyte [13] are strongly consistent data stores that offer distributed ACID transactions. Further examples include Calvin [51], HBase [7], Google's F1 [43], and VoltDB [48], among others. While TAO's query APIs are much simpler, these systems are not read-optimized to suit our needs.

Our solution for stronger isolation guarantees operates in an read-heavy environment and enables read transactions to be processed without significant latency overheads. We navigate a similar tradeoff as Zanzibar [39], which is built on top of Google's strictly serializable Spanner [24] but exposes a weaker consistency model to clients for improved read efficiency and latency. There has also been extensive work on supporting transactions with asynchronous replication. Some systems require applications to identify and isolate operation side effects [30] while others rely on maintaining large dependency sets [32, 33, 36, 46]. RAMP-TAO provides transactional semantics in an asynchronously replicated environment without additional work from the application side and has limited storage overhead.

Isolation models. Many databases offer weak isolation levels, which enable greater concurrency and improved performance [16, 37]. Industrial systems such as BigTable [22], Espresso [41], Manhattan [2], and Voldemort [49] provide Read Uncommitted isolation in which writes to each object are totally ordered, but this order can differ per replica.

To the best of our knowledge, the RAMP protocols [17] are the first work to explicitly address Read Atomic isolation. Subsequent work in [47] presents an protocol for providing atomic visibility for the serverless setting. We adapt RAMP for a read-optimized, geo-distributed environment.

A range of systems have been built to ensure stronger isolation levels at the cost of unavailability and increased latency. Snapshot isolation is offered by some systems [10, 12, 15, 40, 50]. Serializability is another popular choice, found in FaunaDB [9], G-Store [25], H-Store [28], L-Store [31], MDCC [29], Megastore [19], Spanner [24], SLOG [42], and TAPIR [52], among others. Our work shows how we can ensure Read Atomic isolation while maintaining availability and scalability.

Stronger guarantees at Facebook. Facebook has published several studies on the consistency semantics of its systems. Ajoux et. al. identified four fundamental challenges [14] in providing causal consistency at Facebook: (1) integrating across many stateful services, (2) tolerating high query amplification, (3) handling linchpin objects, and (4) providing a net benefit to users. We address all of these issues in our efforts to strengthen isolation guarantees on TAO. We find that our layered approach for transactional semantics allows developers to benefit from the simplicity, convenience, and extended functionality of stronger isolation, without affecting users that do not require these guarantees.

Lu et. al. measured replica consistency on TAO [34]. While several consistency models were explored, this study did not consider transactions. The emergence of the Ent framework and TAO transactions have increased the scope of semantic boundaries. Furthermore, Ent queries can span multiple systems in addition to TAO and make it more difficult to ensure stronger guarantees. FlightTracker [45] is Facebook's approach to distributed consistency but does not address transactions and isolation guarantees. Our work is the first to study transactional anomalies at the application-level and demonstrate how to strengthen isolation guarantees for TAO.

8 CONCLUSION

This paper describes the challenges and importance of offering transactional semantics and stronger isolation guarantees for TAO. We explain why developers need transactions at Facebook and describe the current failure-atomic write transactions on TAO. We then measure potential anomalies that can arise in production from batched reads and demonstrate the need for stronger guarantees in a read transaction API. As our solution, we present the RAMP-TAO protocol, which ensures atomic visibility for a read-optimized, geographically distributed environment. We implement a prototype of this protocol to show its feasibility for production. Our benchmarks illustrate that we can provide high-performance, atomically visible read transactions with little memory overhead. We also give insight into the benefits of an incremental, layering approach to stronger isolation guarantees for large-scale systems.

ACKNOWLEDGMENTS

We thank Aaron Kabcenell, Emmanuel Geay, Scott Pruett, Jim Carrig, and the VLDB reviewers for their insightful feedback; Sam Kline, Matt Pugh, Amin Saeidi, David Goode, Soham Shah, Evan Liang, Zhuo (Gordon) Huang, Dan Lambright, and Robert Cooper for their contributions to the systems in this paper. Audrey Cheng is supported in part by the NSF CISE Expeditions Award CCF-1730628 and gifts from Alibaba Group, Amazon Web Services, Ant Group, CapitalOne, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, Nvidia, Scotiabank, Splunk, and VMware.

REFERENCES

[1] 2013. The Apache Software Foundation Announces Apache™ Cassandra™ v2.0. https://blogs.apache.org/foundation/entry/the_apache_software_foundation_announces44

[2] 2014. Manhattan, our real-time, multi-tenant distributed database for Twitter scale. https://blog.twitter.com/engineering/en_us/a/2014/manhattan-our-real-time-multi-tenant-distributed-database-for-twitter-scale.html

[3] 2014. Under the Hood: Building and open-sourcing fbthrift. https://engineering.fb.com/2014/02/20/open-source/under-the-hood-building-and-open-sourcing-fbthrift/

[4] 2018. ACID Transactions in MongoDB. https://www.mongodb.com/transactions

[5] 2018. Amazon DynamoDB Transactions. https://aws.amazon.com/blogs/aws/new-amazon-dynamodb-transactions/

[6] 2020. MySQL Transactional and Locking Statements. https://dev.mysql.com/doc/refman/8.0/en/sql-transactional-statements.html

[7] 2021. Apache HBase. https://hbase.apache.org/

[8] 2021. etcd. https://etcd.io

[9] 2021. FaunaDB. https://fauna.com/

[10] 2021. Oracle Database Development Guide Release 21. https://docs.oracle.com/en/database/oracle/oracle-database/21/adfns/sql-processing-for-application-developers.html#GUID-912124B7-0F39-47BD-863C-320ED61014E9

[11] 2021. Redis. https://redis.io/

[12] 2021. SAP HANA Reference: SET TRANSACTION. https://help.sap.com/viewer/4fe29514fd584807ac9f2a04f6754767/2.0.05/en-US/20fdf9cb75191014b85aaa9dec841291.html?q=serializable

[13] 2021. Yugabyte DB. https://www.yugabyte.com

[14] Phillipe Ajoux, Nathan Bronson, Sanjeev Kumar, Wyatt Lloyd, and Kaushik Veeraraghavan. 2015. Challenges to Adopting Stronger Consistency at Scale. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. USENIX Association, Kartause Ittingen, Switzerland. https://www.usenix.org/conference/hotos15/workshop-program/presentation/ajoux

[15] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, Vijendra Purohit, Hugh Qu, Chaitanya Sreenivas Ravella, Krystyna Reisteter, Sheetal Shrotri, Dixin Tang, and Vikram Wakade. 2019. Socrates: The New SQL Server in the Cloud. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1743–1756.

[16] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013. Highly Available Transactions: Virtues and Limitations. *Proc. VLDB Endow.* 7, 3, 181–192.

[17] Peter Bailis, Alan Fekete, Joseph M. Hellerstein, Ali Ghodsi, and Ion Stoica. 2014. Scalable Atomic Visibility with RAMP Transactions. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 27–38.

[18] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013. Bolt-on Causal Consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 761–772.

[19] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. 2011. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*. 223–234.

[20] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley.

[21] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. 2013. TAO: Facebook's Distributed Data Store for the Social Graph. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. USENIX Association, San Jose, CA, 49–60.

[22] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2006. Bigtable: A Distributed Storage System for Structured Data. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 06)*. USENIX Association, Seattle, WA.

[23] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. 2008. PNUTS: Yahoo!'s Hosted Data Serving Platform. *Proc. VLDB Endow.* 1, 2, 1277–1288.

[24] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2012. Spanner: Google's Globally-Distributed Database. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX Association, Hollywood, CA, 261–264.

[25] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. 2010. G-Store: A Scalable Data Store for Transactional Multi Key Access in the Cloud. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 163–174.

[26] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-Value Store. *SIGOPS Oper. Syst. Rev.* 41, 6, 205–220.

[27] J. N. Gray. 1978. Notes on data base operating systems. In *Operating Systems: An Advanced Course*, R. Bayer, R. M. Graham, and G. Seegmüller (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 393–481.

[28] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. 2008. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System.

[29] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. 2013. MDCC: Multi-Data Center Consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems* (Prague, Czech Republic) *(EuroSys '13)*. Association for Computing Machinery, New York, NY, USA, 113–126.

[30] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. 2012. Making Geo-Replicated Systems Fast as Possible, Consistent When Necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Hollywood, CA, USA) *(OSDI'12)*. USENIX Association, USA, 265–278.

[31] Qian Lin, Pengfei Chang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Zhengkui Wang. 2016. Towards a Non-2PC Transaction Management in Distributed Database Systems. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1659—-1674.

[32] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2013. Stronger Semantics for Low-Latency Geo-Replicated Storage *(nsdi'13)*. USENIX Association, USA, 313–328.

[33] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2013. Stronger Semantics for Low-Latency Geo-Replicated Storage. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX Association, Lombard, IL, 313–328.

[34] Haonan Lu, Kaushik Veeraraghavan, Philippe Ajoux, Jim Hunt, Yee Jiun Song, Wendy Tobagus, Sanjeev Kumar, and Wyatt Lloyd. 2015. Existential Consistency: Measuring and Understanding Consistency at Facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 295–310.

[35] Yoshinori Matsunobu, Siying Dong, and Herman Lee. 2020. MyRocks: LSM-Tree Database Storage Engine Serving Facebook's Social Graph. *Proc. VLDB Endow.* 13, 12, 3217–3230.

[36] Syed Akbar Mehdi, Cody Littley, Natacha Crooks, Lorenzo Alvisi, Nathan Bronson, and Wyatt Lloyd. 2017. I Can't Believe It's Not Causal! Scalable Causal Consistency with No Slowdown Cascades. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 453–468.

[37] C. Mohan. 2013. History Repeats Itself: Sensible and NonsenSQL Aspects of the NoSQL Hoopla *(EDBT '13)*. Association for Computing Machinery, New York, NY, USA, 11–16.

[38] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. 2013. Scaling Memcache at Facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX Association, Lombard, IL, 385–398.

[39] Ruoming Pang, Ramon Caceres, Mike Burrows, Zhifeng Chen, Pratik Dave, Nathan Germer, Alexander Golynski, Kevin Graney, Nina Kang, Lea Kissner, Jeffrey L. Korn, Abhishek Parmar, Christopher D. Richards, and Mengzhi Wang. 2019. Zanzibar: Google's Consistent, Global Authorization System. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 33–46.

[40] Daniel Peng and Frank Dabek. 2010. Large-Scale Incremental Processing Using Distributed Transactions and Notifications. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Vancouver, BC, Canada) *(OSDI'10)*. USENIX Association, USA, 251–264.

[41] Lin Qiao, Kapil Surlaker, Shirshanka Das, Tom Quiggle, Bob Schulman, Bhaskar Ghosh, Antony Curtis, Oliver Seeliger, Zhen Zhang, Aditya Auradar, Chris Beaver, Gregory Brandt, Mihir Gandhi, Kishore Gopalakrishna, Wai Ip, Swaroop Jgadish, Shi Lu, Alexander Pachev, Aditya Ramesh, Abraham Sebastian, Rupa Shanbhag, Subbu Subramaniam, Yun Sun, Sajid Topiwala, Cuong Tran, Jemiah Westerman, and David Zhang. 2013. On Brewing Fresh Espresso: LinkedIn's Distributed Data Serving Platform. In *Proceedings of the 2013 ACM SIGMOD International*

Conference on Management of Data (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 1135–1146.

[42] Kun Ren, Dennis Li, and Daniel J. Abadi. 2019. SLOG: Serializable, Low-Latency, Geo-Replicated Transactions. *Proc. VLDB Endow.* 12, 11, 1747—-1761.

[43] Bart Samwel, John Cieslewicz, Ben Handy, Jason Govig, Petros Venetis, Chanjun Yang, Keith Peters, Jeff Shute, Daniel Tenedorio, Himani Apte, Felix Weigel, David Wilhite, Jiacheng Yang, Jun Xu, Jiexing Li, Zhan Yuan, Craig Chasseur, Qiang Zeng, Ian Rae, Anurag Biyani, Andrew Harn, Yang Xia, Andrey Gubichev, Amr El-Helw, Orri Erling, Zhepeng Yan, Mohan Yang, Yiqun Wei, Thanh Do, Colin Zheng, Goetz Graefe, Somayeh Sardashti, Ahmed M. Aly, Divy Agrawal, Ashish Gupta, and Shiv Venkataraman. 2018. F1 Query: Declarative Querying at Scale. *Proc. VLDB Endow.* 11, 12, 1835—-1848.

[44] Yogeshwer Sharma, Philippe Ajoux, Petchean Ang, David Callies, Abhishek Choudhary, Laurent Demailly, Thomas Fersch, Liat Atsmon Guz, Andrzej Kotulski, Sachin Kulkarni, Sanjeev Kumar, Harry Li, Jun Li, Evgeniy Makeev, Kowshik Prakasam, Robbert Van Renesse, Sabyasachi Roy, Pratyush Seth, Yee Jiun Song, Benjamin Wester, Kaushik Veeraraghavan, and Peter Xie. 2015. Wormhole: Reliable Pub-Sub to Support Geo-replicated Internet Services. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 351–366.

[45] Xiao Shi, Scott Pruett, Kevin Doherty, Jinyu Han, Dmitri Petrov, Jim Carrig, John Hugg, and Nathan Bronson. 2020. FlightTracker: Consistency across Read-Optimized Online Stores at Facebook. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 407–423.

[46] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. 2011. Transactional Storage for Geo-Replicated Systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal) *(SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 385–400. https://doi.org/10.1145/2043556.2043592

[47] Vikram Sreekanti, Chenggang Wu, Saurav Chhatrapati, Joseph E. Gonzalez, Joseph M. Hellerstein, and Jose M. Faleiro. 2020. A Fault-Tolerance Shim for Serverless Computing. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 15, 15 pages.

[48] Michael Stonebraker and Ariel Weisberg. 2013. The VoltDB Main Memory DBMS. *IEEE Data Eng. Bull.*

[49] Roshan Sumbaly, Jay Kreps, Lei Gao, Alex Feinberg, Chinmay Soman, and Sam Shah. 2012. Serving Large-Scale Batch Computed Data with Project Voldemort. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies* (San Jose, CA) *(FAST'12)*. USENIX Association, USA, 18.

[50] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. 2020. CockroachDB: The Resilient Geo-Distributed SQL Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1493–1509.

[51] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: Fast Distributed Transactions for Partitioned Database Systems *(SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 1–12.

[52] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. 2018. Building Consistent Transactions with Inconsistent Replication. *ACM Trans. Comput. Syst.* 35, 4, Article 12, 37 pages.