FlightTracker: Consistency across Read-Optimized Online Stores at Facebook

Xiao Shi, Scott Pruett, Kevin Doherty, Jinyu Han,
Dmitri Petrov, Jim Carrig, John Hugg, Nathan Bronson
Facebook, Inc.

Abstract

Social media platforms deliver fresh personalized content by performing a large number of reads from an online data store. This store must be optimized for read efficiency, availability, and scalability. Multi-layer caches and asynchronous replication can satisfy these goals, such as in Facebook's graph store TAO, but it is challenging for the resulting system to provide a developer-friendly consistency model. TAO originally provided read-your-writes (RYW) consistency via write-through caching, but scaling challenges with this approach have led us to a new implementation.

This paper introduces FlightTracker, a family of APIs and systems which now manage consistency for online access to Facebook's graph. FlightTracker implicitly provides RYW and can be explicitly used to provide alternative consistency guarantees for special use cases; it enables flexible communication patterns between caches, which we have found important as the number of datacenters increases; it extends the same consistency guarantees to cross-shard indexes and materialized views, allowing us to transparently optimize queries; and it provides a uniform primitive for clients to obtain desired consistency guarantees across a variety of data stores. FlightTracker delivers these advantages while preserving the efficiency, latency, and availability benefits of asynchronous replication for the underlying systems, managing consistency for billions of users and more than 10^{15} queries per day.

1 Introduction

Social media platforms deliver fresh and customized aggregation of content. This feature combination makes it ineffective to aggregate ahead of time; instead each application-level web request at Facebook may issue hundreds or thousands of queries to our graph store TAO [20] to render a single response. This high query amplification means that data store reads must be efficient, low-latency, and highly available. At Facebook, we have addressed this challenge with an asynchronously coupled federation of caches, database replicas, and customized indexes that model social media data and metadata as a graph. While this read-optimized ecosystem achieves high performance, it is challenging to provide an intuitive and uniform consistency model to developers.

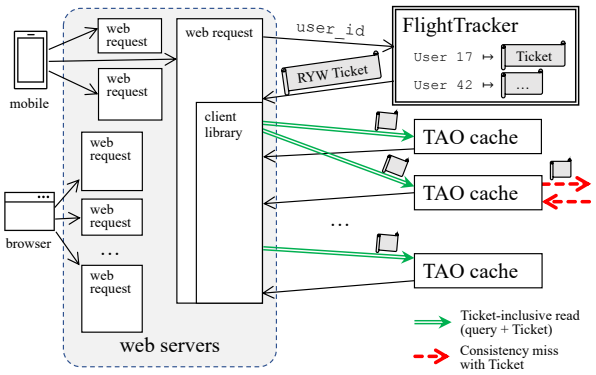

Figure 1: Web request flow with FlightTracker.

FlightTracker is our solution for managing RYW consistency for online access to the social graph at Facebook. It preserves the read efficiency, hot spot tolerance, and high availability of eventual consistency while providing RYW consistency. FlightTracker offers a uniform notion of an end user session that spans many stateful services and can be extended to new data stores without architecture changes.

FlightTracker consists of a family of APIs and a metadata service. Building on write-set tracking techniques [28, 40, 41, 51] and CRDTs [18, 50], the FlightTracker service accumulates the metadata of a user's recent writes and exposes the metadata as a data type we call a *Ticket*. Web requests fetch the user's Ticket once, as soon as the user is identified (see Figure 1). This Ticket is automatically attached on all subsequent queries to the social graph from the web request.

We use a variety of system-specific strategies to ensure that every write identified by the Ticket is reflected in query results. For example, our strategy for caches is to ignore cache entries that may be stale compared to writes in the Ticket; we refer to the resulting cache miss as a *consistency miss*. Systems can propagate Tickets recursively when they need to fetch data from another component while processing a query.

FlightTracker has been in production since 2016. It provides RYW consistency for billions of users and 10^{15} data store queries per day. For the majority of Facebook's internal applications and developers, FlightTracker is automatic and hidden. Some call sites and higher-layer infrastructure components explicitly manipulate Tickets to strengthen the consistency level. FlightTracker's loosely coupled design has allowed us to incrementally roll out support to two caching systems, three indexing systems, and two databases. It preserves the efficiency, latency, and availability that these data stores would offer under eventual consistency.

Overall, this paper makes five contributions:

- We summarize challenges Facebook encountered when relying on write-through caching for RYW in TAO, a read-optimized geo-replicated graph store (§ 2).
- We present the Ticket abstraction, which encapsulates the system-specific details of write sets in an extensible manner across service boundaries (§ 4).
- We show how we store and exchange Tickets in the FlightTracker service to provide RYW consistency (§ 3 and § 5) or explicitly satisfy alternative consistency requirements for select use cases (§ 7) while tolerating hot spots (§ 6.5).
- We explain a variety of strategies we used to implement Ticket-inclusive reads in query-serving systems (§ 6), including ones for simple caches and global indexes with complex update pipelines (§ 6.3).
- We evaluate FlightTracker in our production environment, demonstrating that it preserves the useful properties of the underlying read-optimized stores (§ 8), and share some lessons learned (§ 8.5).

2 Motivation

TAO is a read-optimized data store that provides access to the social graph at Facebook [20]. It is implemented using two layers of caches in front of a geo-replicated database. TAO originally relied on write-through caching for consistency. This technique provided RYW on top of eventual consistency, while preserving the read efficiency and hot spot tolerance of the system, since it allowed most queries to be served from a nearby L1 cache server.

As Facebook grew, we found that we needed a better approach to consistency. TAO's original write-through strategy relied on the use of a fixed communication pattern: users were made sticky to a single L1 cache cluster by the load balancers, inter-cluster communication was limited to traversing a fixed tree, and writes were proxied along the same tree traversal chain that would be followed on a read miss. RYW would be violated if any of the following were true: user requests were routed to another cluster of web servers; the mapping from web server cluster to L1 cache cluster was changed; queries were failed over to a stale replica; cache contents were lost

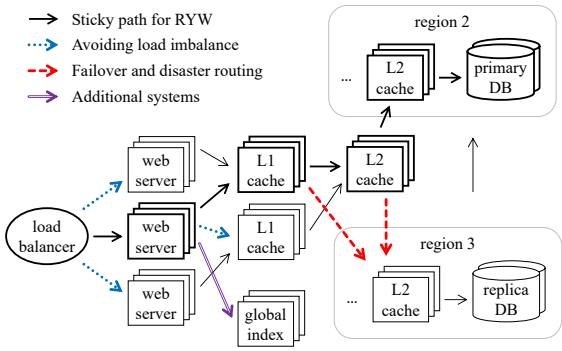

Figure 2: RYW via write-through caching excludes many useful inter-cluster communication patterns.

before asynchronous replication occurred; or any query was served by a data store other than TAO.

2.1 Scaling challenges

As TAO's footprint grew, we found it increasingly problematic to rely on a fixed communication topology. In fact, each of the conditions required for write-through RYW became harder to satisfy over time. As cross-cluster networking improved, we moved away from pairing and collocating L1 caches with web server clusters, reducing the number of L1 cache replicas per region. This reduced the number of cached copies of data, but it required fractional or dynamic assignment of web server traffic to L1 cache clusters to get reasonable balance. Switching from cluster-sticky to region-sticky user routing improved the load distribution of both the web server clusters and TAO. As the number of geographic regions grew, we started to deploy TAO in some datacenters without a local database replica, routing cache misses to the closest neighboring region. If we were restricted to a tree topology for miss routing and cache invalidation streams, the outage of one database replica would affect multiple regions. Figure 2 shows some of the desirable communication patterns we encountered that break the write-through consistency model. The dotted and dashed arrows show read requests that potentially violate RYW consistency without FlightTracker.

Another recurring issue was queries that needed cross-cluster or global write visibility. TAO marks these queries *critical*, routing them to the L2 cache in the region holding database primaries, near the base of the communication tree [20]. This strategy has latency and availability drawbacks. It is also not tolerant of spiky workloads.

2.2 Cross-system consistency

As we encountered challenges scaling TAO's write-through approach to consistency, the social graph ecosystem expanded. Application developers moved from directly accessing TAO to using a query language that makes it easy to express multi-hop and attribute-filtering predicates over the graph. This layer of indirection allowed us to build additional systems tailored to a subset of the Facebook query workload.

Some application queries involve many round trips when mapped onto TAO's simple API and transfer a lot of data that the client immediately discards. Global secondary indexes can optimize the communication pattern of these queries, but it is only safe to transparently or retroactively optimize execution using indexes if the index stores have the same semantics as TAO [34]. Our indexing systems are loosely coupled, updated by asynchronous pipelines that reshard, transform, and filter. Loose coupling enables separate development and deployment, but it limits the consistency implementation strategies. Most indexes are sharded differently than TAO, so even if we used a more monolithic design, they could not participate synchronously in the write path without reducing availability and increasing tail latency [17, 54].

Another side effect of moving to the application-level query language for the social graph was that it became easier to use alternate database technologies as the system of record for parts of the social graph, such as for data types that experience high write rates or limited lifetimes. These systems also experience the same consistency challenges as TAO.

Ajoux et al. [10] previously identified four fundamental challenges to providing causal consistency in Facebook's social media platform: integrating across many stateful services, tolerating high query amplification, handling linchpin objects (i.e., hot spots), and providing a net benefit for users. Our experience has been that these challenges also arise when providing RYW consistency and that the most difficult hurdle is producing a design that addresses all of them simultaneously.

2.3 Why read-your-writes?

Consistency models for data stores make guarantees about what writes should be visible to a read. Application developers use these guarantees to reason about the correctness of the entire system. Strong models like linearizability [32] or causal consistency [9] generally provide a simpler experience and mental model for developers, but they constrain the implementation.

Providing consistency guarantees for read-optimized systems boils down to implementing a staleness check to determine whether a cache or replica can serve a read query with its local data. This staleness check must be: (1) local, avoiding network communication in most cases; (2) highly granular, so that few queries result in extra work due to false positives from the checks; and (3) conducive to incremental repair, so that the extra work to find fresh data can be reused for subsequent queries. Importantly, staleness checks are needed for single-replica reads even in systems that use synchronous quorum writes. For example, Raft followers [43] or Paxos acceptors [38] might have no knowledge of a write committed by the leader if they were not part of the commit quorum.

Logical and physical timestamps, such as Hybrid logical clocks [36], Spanner's TrueTime [26], and Occult's compressed vector clocks [41], provide a simple and scalable way to check for staleness—the local data is sufficiently fresh if its timestamp is higher than the desired read timestamp. Unfortunately, these approaches are neither granular nor conducive to incremental repair. If the local store is 10 seconds behind the desired read timestamp, for example, it cannot service any queries until it has processed all of the missing writes.

For our workload, it is important that we can serve most queries locally, even if the local replicas are a few seconds stale. This led us to reject consistency levels in which all writes (linearizability) or most writes (causal consistency) missing from a stale replica need to be visible. In contrast, RYW allows us to utilize a stale replica by adding fresh versions of only a limited set of writes ("your" writes). We also rejected weaker models like bounded staleness that do not guarantee that a user sees their own writes, which are difficult to use correctly for an interactive application [54].[1]

Our experience at Facebook has been that the simple RYW consistency model [51] is a reasonable default for application developers and our end users, with an extension: we want to extend the concept of a session to end users.

User-centric sessions: Our desire to implement user-centric session RYW guarantees means that we experience intra-session concurrency at several levels, as shown in Figure 1: a single web request issues TAO reads and writes in parallel; a single browser or mobile app has many web requests in flight at once; and a user may even be accessing Facebook simultaneously from multiple devices.

The original definition of RYW session guarantees [51] implies that reads and writes within a session are totally ordered and that this intra-session order coincides with the physical order of those operations, as in linearizability. As a result, while the theoretical definition does not require a single-threaded session, implementations limit sessions to a single client, writer, session manager, or server [19, 28, 42, 52].

However, our observation is that application developers do not expect concurrent web requests to communicate with each other. A common mental model is that concurrent requests execute in a random order and possibly interleave with each other, so visibility from one request to the next is only assured if one request finishes before the other starts. An application developer's intuition is that the first moment data is guaranteed to be available is when the write acknowledgement is received by the local program that issued the web request.

This observation led us to the following relaxation of the RYW guarantee, which is what FlightTracker provides:

A read to the social graph will observe all writes done by the same end user in previously completed web requests or in the same web request.

This definition gives us much-needed flexibility for handling intra-session concurrency. Note that as in the original RYW definition, a read may "observe" a write by returning an even newer version of the data.

[1] We think providing both RYW and bounded staleness is an interesting and feasible model, even for our read-optimized environment.

3 FlightTracker

The main idea of FlightTracker is to decompose the problem of RYW consistency into three parts: (1) the Ticket abstraction, a flexible and extensible way of representing write sets across independently developed systems and APIs; (2) the FlightTracker service, generic infrastructure queried once per web request to get a user's recent write metadata; and (3) Ticket-inclusive reads, system-specific mechanisms to ensure that the specified writes are reflected in query results.

Our goal for FlightTracker is to preserve the communication patterns that benefit eventually consistent read-optimized stores. FlightTracker piggybacks on existing messages in these data stores. Most read queries can be served by a single local RPC, maintaining high efficiency and low latency. FlightTracker does not restrict where data stores send read RPCs, which allows them to leverage per-query retry and failover for high availability. FlightTracker supports the aggressive multi-level caching that data stores use to tolerate hot spots. Most of the work to ensure writes become visible to reads is handled by asynchronous pipelines, which retains the desired isolation and loose coupling of the underlying data stores. This piggybacking approach has also made it feasible to incrementally add FlightTracker support to existing mature systems with low overhead.

Figure 3 shows the API extensions a data store needs to implement to integrate with FlightTracker. On a successful write, a data store returns a Ticket identifying the write alongside the result; read queries take a Ticket parameter and guarantee any relevant writes in the Ticket will be reflected in the result.

3.1 An example

Consider a hypothetical social media product using TAO's graph model of versioned nodes and edges, with user nodes, media nodes, edges when a user has enjoyed a particular media instance, and edges when a user trusts another's tastes. Let's say Alice enjoys Mozart's *Requiem*; Bob recently indicated he trusts Alice's taste in art and then expanded his trust in Alice to include music. The resulting subgraph is shown in Figure 5 and Bob's recent writes to TAO would be:

$$\mathit{WriteEdge}(\langle 17, \texttt{TRUSTS}, 42\rangle \mapsto \{\texttt{"art"}\}) \quad (1)$$
$$\mathit{WriteEdge}(\langle 42, \texttt{TRUSTED\_BY}, 17\rangle \mapsto \{\texttt{"art"}\}) \quad (2)$$
$$\mathit{WriteEdge}(\langle 17, \texttt{TRUSTS}, 42\rangle \mapsto \{\texttt{"art"}, \texttt{"music"}\}) \quad (3)$$
$$\mathit{WriteEdge}(\langle 42, \texttt{TRUSTED\_BY}, 17\rangle \mapsto \{\texttt{"art"}, \texttt{"m...}\}) \quad (4)$$

To get RYW consistency, we simply need to ensure that Bob's subsequent data store queries include the effects of these writes. We do this by computing Bob's recent write set once per web request, attaching it to all of his queries, and then making sure the data stores reflect attached writes in the query results.

3.2 Tickets

We store write metadata in a data type we call a Ticket. Metadata to identify a write includes information like the transaction ID and the resulting node or edge version but does not include the data itself. If W_i is metadata that identifies Bob's i-th write above, we might have:

$$W_3 = [\mathit{key} \mapsto \langle 17, \texttt{TRUSTS}, 42\rangle, \\ \mathit{op} \mapsto \mathit{WriteEdge},\ v \mapsto 2,\ \mathit{txn\_id} \mapsto 8980]$$
$$W_4 = [\mathit{key} \mapsto \langle 42, \texttt{TRUSTED\_BY}, 17\rangle,\ \ldots,\ \mathit{txn\_id} \mapsto 8985]$$

Bob's Ticket would then be $\{W_1 \ldots W_4\}$.

As write sets, Tickets can be joined via set union. Moreover, Tickets are handled and passed around between many independently deployed systems; therefore, they need to be encapsulated, extensible, and forward- and backward-compatible. Inside a Ticket, writes can be enumerated or represented using a low-water mark that implicitly includes all preceding writes. § 4 describes Ticket contents, semantics, and implementation.

```
pair<Result, Ticket> write(...); // returns metadata
Result read(..., Ticket); // Ticket-inclusive read
```

Figure 3: API extensions data stores expose to participate in the FlightTracker ecosystem.

```
void appendWrite(SessionId, Ticket); // FT write
Ticket getMergedWrites(SessionId); // FT read
```

Figure 4: The API of the FlightTracker service.

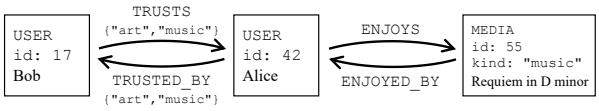

Figure 5: Subgraph for a hypothetical application.

3.3 The FlightTracker service

Bob's logical user session spans many web requests, so we need to store metadata for his recent writes elsewhere. To that end, we built the FlightTracker service with an API resembling a hash map of user IDs to recent writes (Figure 4). For example, Bob's entry will be $17 \mapsto \{W_1, W_2, W_3, W_4\}$. The client library calls `appendWrite` immediately after a successful write to the data store before acknowledging success to the application; `getMergedWrites` returns the recent writes for a particular user.

Figure 1 shows an RPC pattern that might occur as Bob browses the music portion of the site. As soon as the web request identifies Bob as the logged-in user, it fetches his RYW Ticket from FlightTracker by calling `getMergedWrites(17)` and puts it into the web request context. When Bob performs a write, the client library joins its metadata into the Ticket in the web request context and also uses `appendWrite` to immediately send the write metadata to FlightTracker. The client library implicitly attaches the Ticket from the web request context to every read query. A single web request performs many such queries, offering ample opportunity to amortize the initial Ticket fetch. Most developers do not explicitly observe or manipulate Tickets.

3.4 Ticket-inclusive reads

It is the responsibility of the data store clients to attach a Ticket that ensures RYW to each query, and it is the responsibility of each query-serving component to ensure that all of the writes in the Ticket are included in a query result.

Our applications do not expect to have exclusive access to the social graph or to read from snapshots; reads are always allowed to return data that is fresher than expected. In Ticket-inclusive reads, a Ticket specifies a lower bound on writes that should be visible. A Ticket that encodes a superset of another can always be safely substituted at read time, as anything made visible by the superset Ticket might have been visible anyway as part of normal asynchronous replication.

Cache queries: After getting his RYW Ticket, Bob's web request performs two queries to TAO's cache. The simplicity of the TAO API makes it straightforward for the cache to validate the freshness of its cache content—a TAO replica compares the versions of the data in question against the versions specified in the Ticket. For example, if the request is reading a list of all of the users that Bob trusts ($GetEdges(\langle 17, \texttt{TRUSTS}, * \rangle)$), then W_3 implies the edge to Alice must be present with version ≥ 2.

In Figure 1, the first TAO query was a cache hit unaffected by the Ticket. This is the common case. The second TAO query shows a *consistency miss*, where the local cache contents are stale. In this case, the cache goes upstream and merges the fresh edge into the local list before responding. Note that the upstream query has the same Ticket attached, which recursively ensures visibility of Bob's recent writes.

Index queries: If Bob is browsing the song with ID 55, we would like to display Bob's trusted users who also enjoy it. This involves finding all x where $\langle 17, \texttt{TRUSTS["music"]}, x \rangle \wedge \langle 55, \texttt{ENJOYED\_BY}, x \rangle$. This two-hop query is not well suited to TAO, because both the `TRUSTS` and `ENJOYED_BY` edge lists may be too large to fully cache. We can optimize this type of query by materializing a global secondary index. Specifically, we might use a list-intersection index with edge lists for `TRUSTS` edges that include `"music"` and `ENJOYED_BY` edges from `"music"` `MEDIA` nodes.

An index leaf (read server) does not have enough information to accurately identify missing writes, because writes that are filtered by the update pipeline will never arrive. For example, W_1 can be filtered upstream because it is not a `TRUSTS` edge that includes `"music"` and thus does not change any materialized lists. Without extra information, an index leaf will consider W_1 missing forever. Any such index leaf cannot satisfy a Ticket-inclusive read with Bob's Ticket with W_1 in it. We solve this problem by tracking the delivery information of recent writes, including the recent routing and filtering choices of the update pipeline as well as the delivery status to the index leaves. This FlightTracker-ReverseIndex (FT-RI) component builds an index of recent writes to the actions taken by the index update pipeline (§ 6.3). Queryable using the metadata present in Tickets, the delivery information is used by the index client library to determine whether an index read result is fresh enough. If stale, the client library uses strategies such as read repair or retry to obtain a fresh result.

§ 6 describes our full range of strategies to ensure results of Ticket-inclusive reads reflect all writes in a Ticket.

4 Ticket details

A Ticket is a set of write metadata. We use Tickets to identify writes to the social graph regardless of where the writes are committed. Tickets allow generic infrastructure to track and identify writes across many independently deployed systems, while letting databases convey system-specific details. For clarity, we refer to the systems that persist the normalized data and generate the write metadata as "databases," as opposed to other data stores such as caches and indexes which mostly serve reads and proxy writes.

A Ticket is implemented as a union of custom per-database representations. On a write, a single-database Ticket is minted with only metadata for the newly committed write (Figure 3). It can then be joined with other Tickets or otherwise used by FlightTracker and custom applications, producing Tickets that may contain writes from multiple databases.

The encapsulation of Tickets and the semantics of Ticket-inclusive reads together give us great flexibility in the Ticket implementation. Since Ticket-inclusive reads interpret Tickets as lower bounds, read results containing additional writes or fresher writes than exactly encoded in the Ticket are unsurprising to the applications. Furthermore, thanks to encapsulation, applications cannot examine the exact content or representation of a Ticket, which means we can always safely include additional writes inside a Ticket. We leverage this flexibility in Ticket compaction (§ 4.2) and Ticket replication in the FlightTracker service (§ 5).

4.1 Identifying a set of writes

The naive strategy of identifying writes by globally unique IDs is easy to implement but difficult to use—read-serving data stores must keep track of all write IDs to determine whether the local replica is sufficiently up-to-date. Assigning a total write order allows systems to identify writes by their ordinal positions. However, ordering implies synchronization via communication or via timed-wait [26]. To preserve the efficiency benefits of asynchronous replication, databases often opt for limited-scope ordering. In our experiences, there are three natural scopes:

- Per-key: Any strictly monotonically increasing value can be combined with a key to identify a particular change to that row or object. This version need not be contiguous—a monotonically increasing timestamp would also suffice.
- Per-shard: Many databases totally order all writes to a particular shard, but give no order guarantees between shards.

- Global: Some systems [26,53] offer total order of all writes globally. HLCs or known-accuracy clocks can also order writes across shards and database types.

All of the databases supported by FlightTracker have the following properties which allow us to further simplify our assumptions. Our databases expose a versioned key-object model. They are sharded and maintain a total order for all writes within a shard; furthermore, writes in a single shard are replicated in the same order. The commit-time information (such as commit timestamps or transaction IDs) can thus specify a contiguous prefix of that shard and serve as a low-water mark to determine the replication state of a data store.

Most fields in a Ticket can be and are empty. When new Tickets are minted, they can include any information known at commit time, such as per-key versions, commit timestamps, or transaction IDs. While not necessary, including more metadata allows flexibility in interpreting and using the Ticket. Figure 6 shows an example Ticket structure with two databases.

For a data store or client to determine whether a read result is sufficiently up-to-date, this metadata needs to be accessible on reads, which means that it should be stored alongside the data. While this overhead appears non-trivial (e.g., up to 8-bytes per key), our databases already persist these versioning primitives, so the additional cost is negligible.

Although we describe the Ticket abstraction in the context of databases at Facebook, it is applicable and extensible to other databases. Many natively support and store per-key versioning primitives, such as the `rowversion` of Azure SQL Database [13]. Per-shard or global-scope ordering also often underlies modern databases, where writes can be identified by sequence numbers or timestamps stored alongside the client data (e.g., `zxid` for ZooKeeper [33], `hlc` in CockroachDB [2], `offset` in Kafka [35], `LSN` in LogDevice [4]).

4.2 Ticket joining and compaction

Two important operations on Tickets are joining and compaction: joining combines Tickets and compaction reduces their space footprint. Both are local operations with no need for RPCs or information outside the input Tickets.

The *join* operation, which is essentially set union, produces a Ticket that is a superset of all the inputs. It is the primary API for Tickets. For example, data store clients can join Tickets to combine metadata from multiple shards or multiple databases; the FlightTracker service joins Tickets to accumulate per-user recent writes (§ 5); select applications join Tickets to express additional constraints for their reads (§ 7).

Compaction helps Tickets overcome the scaling limit of write set tracking techniques. The idea is straightforward. Tickets represent writes that should be visible, i.e., a kind of lower bound; we can raise the lower bound in exchange for a more compact representation. Intuitively, doing Ticket-inclusive reads with the resulting Ticket makes their constraints equally or more stringent.

Formally, Tickets are CRDTs [50] and the compaction techniques we use are CRDT *inflation* operations [18]: the per-scope ordering and subset-superset relation define the $\leq$ partial order. Ticket inflation produces a Ticket that is $\geq$ the input Ticket according to this order. The resulting Ticket may need fewer bytes to represent. Not all inflation reduces Ticket size but the three types of inflation we use below do:

```
struct RepForDatabaseA {
  map<WriteKey, tuple<Version, TxnId, Timestamp>> perKeyMap;
  map<ShardId, pair<TxnId, Timestamp>> perShardMap;
};
struct RepForDatabaseB {
  map<WriteKey, tuple<Version, Hlc>> perKeyMap;
  map<ShardId, Hlc> perShardMap;
};
struct Ticket {
  RepForDatabaseA repA;
  RepForDatabaseB repB;
  Timestamp globalTs;
};
```

Figure 6: Tickets represent the union of the writes identified by each field.

Per-scope compaction: Keeping the highest version for each key and the highest transaction ID for each shard lets us discard metadata with older versions or transaction IDs. This compaction is performed during every join.

Cross-scope compaction: Some databases have static shard assignments and include both per-key versions and per-shard transaction IDs in the Tickets (such as `DatabaseA` in Figure 6). Replacing per-key metadata with a per-shard transaction ID can greatly reduce the Ticket size, especially for shards with many individual writes. Suppose the edges from the example in § 3.1 among many other writes are all on shard X. We can then compact a Ticket $T_1 = \{W_3, W_4, \ldots, W_{100}\}$ to $T_2 = \{shard_X : txn\_id \mapsto 8985\}$.

Cross-scope compaction offers us a tradeoff between the Ticket size and the cost of serving the Ticket-inclusive read. Since a compacted Ticket semantically encodes more writes, the read is less likely to be served locally. E.g., T_2 requires all writes on shard X with $txn\_id \leq 8985$ to be replicated. Thus, this type of compaction is performed heuristically and sparingly in the FlightTracker service.

Global compaction: We can inflate a Ticket into a global-scope timestamp to represent all writes that have earlier timestamps. Global compaction only happens in the FlightTracker service for writes older than 60 seconds, since we assume the replication lag of our data stores plus clock skews are much smaller. Given the long threshold, global compaction does not have to be exact—older writes do not need to be removed from a Ticket immediately since they purely reduce Ticket size. Thus, the timestamps used for compaction could either be the logical commit timestamp generated by the database or the physical timestamp generated by the data store after the write completes. Global compaction lets FlightTracker store only 60s of data, which greatly reduces its working set.

Additionally, we define a Ticket-inclusive read with an

empty Ticket as implicitly encoding the constraint of returning data no more than 60 seconds stale relative to the current physical timestamp of the replica serving the read. This way, in most cases, we avoid the need to pass around Tickets containing only a single old global timestamp.

4.3 Physical representation

As a cross-system primitive, Ticket presents a number of interesting software-engineering challenges. Ticket-handling code runs inside systems with widely varying deployment frequencies, so Ticket must be both forward- and backward-**compatible**. Ticket **encapsulates** implementation-specific details from multiple systems, but because clients can join Tickets, it cannot leave encoding and decoding up to the data stores. Ticket must be **extensible** and **loosely coupled**, allowing metadata for new systems to be added without affecting existing systems. Ticket must also be **efficient** enough to use on each query in our read-heavy environment.

We address some of the above challenges by using Thrift [8], a serialization format originally designed for efficient and portable RPC. We define the Ticket data structure using the Thrift interface definition language (IDL).

Compatibility: Thrift handles the majority of forward- and backward-compatibility issues, as unknown fields from future versions are silently skipped. Care must still be taken as we introduce new metadata fields to existing systems in the Ticket. For example, if two writes with the same key and timestamp are differentiated by a transaction ID, older code unaware of transaction IDs may be surprised to see a duplicated write.

Serialization: We currently support two serialization formats, identified by the prefix. The default is an LZ4-compression [25] of the Thrift Compact encoding. This is used across all RPC boundaries, making it easy to tunnel Tickets through other systems. The second is the Thrift JSON encoding for readability in debugging and logging.

Encapsulation: A Ticket's internal struct is accessible to systems that mint new Tickets or perform Ticket-inclusive reads. Clients that need not inspect Tickets can treat them as opaque tokens. We provide language bindings and utility functions for code that needs to examine the Ticket internals.

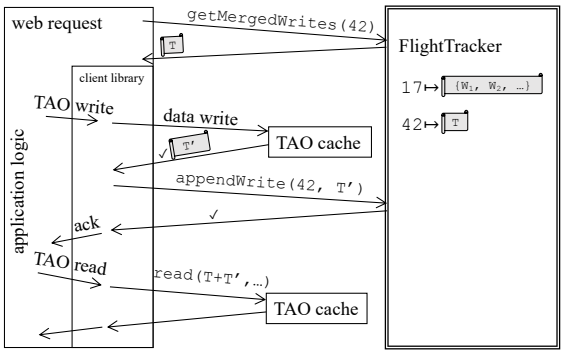

Figure 7: Web request flow with the FlightTracker service.

We chose the name *Ticket* to minimize assumptions developers would make about its semantics. Infrastructure engineers often conflate a visibility guarantee on a specific write (Transaction 8980) with a guarantee on a contiguous prefix (Transactions 1…8980); a new name reduces this tendency.

Extensibility and loose coupling: As shown in Figure 6, each database can customize its own representation. Extending the Ticket structure to support an additional database boils down to adding a field in the main struct and updating the join function. Loose coupling between databases is provided by using different fields for each.

5 FlightTracker service implementation

The API of the stateful FlightTracker service (Figure 4) is extremely simple, consisting of just two operations. As shown in Figure 7, a web request calls `getMergedWrites(user)` at the beginning to get the user session's RYW Ticket; the client library call `appendWrite(user, ticket)` after a database write with the newly minted Ticket and only acknowledge the write to the application if *both* the data store write and `appendWrite` succeed. To reduce ambiguity, we use "data writes" to refer to data store operations and use "metadata writes" to refer to FlightTracker operations.

FlightTracker has the following requirements:

- High throughput: FlightTracker is subject to the full write throughput of all the underlying data stores, since every data write results in a metadata write to FlightTracker. Its effective replication factor is lower than a globally replicated store like TAO, because most writes are only stored in the writing user's region. Its read throughput is proportional to the number of web requests.
- Low latency: Data writes are not acknowledged until their metadata is recorded in FlightTracker, so FlightTracker adds to application-visible write latency.
- High availability: Unavailability of FlightTracker implies loss of availability or loss of RYW consistency for clients. The decoupled nature of FlightTracker allows us to let some use cases fail open (available but inconsistent) while others fail closed (unavailable).
- Durability: A Ticket passed to `appendWrite` should be included in `getMergedWrites` even when there are machine failures. FlightTracker uses a single-round quorum protocol that does not provide atomicity because it is okay for a failed or in-progress `appendWrite` call to be visible.

The working set of FlightTracker is relatively small, as FlightTracker compacts Tickets as it merges them (§ 4.2). Put more practically, if queries are only routed to replicas that are at most 60s stale, then write metadata older than 60s are safe to compact away. Our data stores track the staleness of their own replication streams and a vast majority (>99.99%) of the servers are no more than 60s stale.

5.1 Replication

We implement FlightTracker as a quorum-based store. We statically determine the number of replicas N; the FlightTracker client broadcasts an `appendWrite` to all replicas and considers it successful when W replicas acknowledge the write; a `getMergedWrites` query contacts R replicas and joins the retrieved Tickets.

This plain *single-round* quorum protocol with $R+W>N$ is sufficient to provide the desired correctness guarantee for FlightTracker. A previously appended Ticket will be returned by at least one replica in R, since $R+W>N$ guarantees overlap between the read and write quorums. Merging the read results via `join()` from all R replicas ensures durability: the Ticket will be included in the final read result.

FlightTracker does not need to guarantee atomicity. Recall that given Ticket encapsulation and how Tickets are used as lower bounds, we can safely include additional write metadata in Tickets without violating the overall RYW consistency (§ 4). If a metadata write fails to reach W FlightTracker replicas or is still in progress, FlightTracker *can* safely include it in the result of `getMergedWrites`. Moreover, if a data write succeeds but its metadata write to FlightTracker fails, we consider this write an "unacknowledged success," i.e., the data store client errors out the data write to the application. Application developers do not expect to see failed data writes but know how to handle them if they do show up. Since many of Facebook's applications are built on eventually consistent stores, applications are used to reading fresher writes (e.g., from other applications). Thus, unacknowledged successes are acceptable as long as they are infrequent.

For example, suppose we have a FlightTracker deployment with $N=W=3$ and $R=1$; data writes for W_5 and W_6 completed, but the metadata write for W_5 failed and the metadata write for W_6 is in progress. The state of the three FlightTracker replicas is $\{W_5,W_6\},\{W_6\},\{\}$. A first metadata read could return a Ticket of $\{W_5,W_6\}$ and a subsequent metadata read could return $\{\}$. This is permitted: W_6 has not completed so RYW does not apply yet; W_5, while written in the database, has returned an error to the application, thus the application should have no expectation of visibility either way.

FlightTracker's default setup is a region-local quorum, as Facebook pins logged-in users to a region. We leverage the regional placement to ensure low latency for FlightTracker accesses. Since FlightTracker has a small working set, we choose to store everything in memory and adjust N for redundancy. Empirically, we've found $N=3$ offers a good trade-off between low latency and sufficient redundancy (§ 8).

We statically map user IDs onto logical shards, which are dynamically placed within each FlightTracker replica. Shard placement is aware of load-balancing and covers failure detection. Some use cases use non-user session IDs and cross-region quorums (§ 7.3).

5.2 Failure tolerance

Machine failure is the most common failure that must be handled. Our strategy for this also handles network issues and gaps in coverage during shard movements. We leverage the global compaction bound to restore resiliency after a FlightTracker machine gets a new shard assignment or dynamic shard movement. FlightTracker "warms up" in the first 60 seconds after a shard comes online by accepting all writes but not serving reads. Rejected reads are retried on warm replicas.

5.3 Fail closed vs. fail open

One of the challenges identified by Ajoux et al. [10] was ensuring consistency mechanisms provide a net user benefit. Some applications would prefer to continue a web request even if `getMergedWrites` fails, for example. Given that we have the option to cleanly fail open on a per-query basis, it is difficult to argue for a uniform fail-closed policy. If FlightTracker is completely reliable, there will be vanishingly few inconsistencies even with a fail-open policy, so there is no benefit to fail-closed; on the other hand, if FlightTracker is not completely reliable, then use cases that prefer availability will be harmed by fail-closed. A future option would be to rate-limit fail-closed for those use cases and escalate all fail-open potential RYW violations to an engineer.

Since an error is reported to the application when a data write succeeds but the corresponding `appendWrite` fails, FlightTracker write availability caps the data store availability. FlightTracker is in-memory and region-local, so it has much higher write availability (§ 8) than our underlying persistent data stores; it has a minimal impact on application-visible write availability. Although less important now, the option to fail open was crucial to reducing risk during early rollout.

6 Ticket-inclusive reads

Once FlightTracker has attached a Ticket to a query, it is the responsibility of the data store to ensure that every write identified by the Ticket is reflected in the query result.

The general pattern for implementing Ticket-inclusive reads is that the data store (client or server) filters the writes in the Ticket for relevancy, then checks against its local state to see if they have already been applied. Frequently, the writes in a Ticket are irrelevant to the read (e.g., a write to a `MEDIA` node is irrelevant to reading Alice's `TRUSTED` users) or have been replicated and included locally, in which case the Ticket does not change the result and the read can be served locally.

In the uncommon case that some writes are possibly relevant but missing, i.e., the local data is *possibly* stale, the data store uses a more expensive non-local action to fix the query result. The specific strategies for each of those steps depend on query semantics, the way in which writes are encoded in the Ticket, and what information is locally available.

6.1 Filtering by relevance

Filtering works best for the most granular write representations such as per-key versions. In contrast, timestamps define a write set that includes a contiguous prefix of the history of all data systems at Facebook, which can never be filtered.

For database-specific encodings, a coarse level of filtering happens when the data store ignores writes from other databases in the Ticket. It is also fast and easy to filter by static information in the Ticket or in data store configs, such as TAO object types (e.g., `USER` or `ENJOYS`) or database tables. For Ticket representations that contain keys, we can further filter writes based on query parameters such as the desired node ID. This is highly effective for point queries and simple range queries like TAO's and works for some indexes.

Type and query parameter filtering can be done on the client, which avoids the need to even include a Ticket on most queries. We refer to this operation as *cropping* and have integrated it in the client libraries of all data stores we support.

6.2 Checking inclusion

The systems that incorporate FlightTracker provide eventual consistency on their own, mostly using asynchronous replication. The large majority of writes are delivered with low latency, so most writes are included at the check time.

For database replicas and caches sharded by key, we replicate in write order (i.e., in per-shard *txn_id* order). The replication stream pointer is a compact way of identifying the set of writes included in the local store. If the latest replicated record was 8983, for example, then the write W_3 with $txn\_id \mapsto 8980$ is included but W_4 with $txn\_id \mapsto 8985$ is not.

Cache misses can fetch values from ahead of the replication pointer, so a single low-water mark is not sufficient for high-granularity inclusion checks. TAO maintains a $key \rightarrow txn\_id_{safe}$ mapping that identifies when a particular cache entry is known to include writes newer than the local low-water mark. txn_id_{safe} records the replication position of the upstream source when it serviced the cache miss, not necessarily the transaction at which the key was updated. For example, if a cache with low-water mark 8983 took a cache miss and fetched the edge in W_4 from a database replicated up to 9000, it will have a cache entry with $txn\_id_{safe} \mapsto 9000$ for the edge; the cache is now able to serve point reads to the edge locally if the Ticket has W_4 or even $\{shard_X : txn\_id \mapsto 9000\}$. This exception map is essential to ensuring that Ticket-inclusive queries can still be cache hits.

6.3 Relevance and inclusion for global indexes

Both relevance and inclusion checking are much more challenging for global indexes. An index that lets us find media nodes by their name, for example, will be partitioned using a mutable data attribute rather than by the node's key.

While in this case it would be feasible to include the indexed attribute in the Ticket, we avoid this approach. It bloats the Ticket without solving the problem for all indexes, because the indexed attributes might be from adjacent nodes or edges in the graph. It requires writers to be aware of all index schemas and cannot scale to handle fan-in cases, where a single write affects a large number of index rows. In the example in § 3.4 where we want to answer queries such as "return a list of trusted users who also enjoy a particular song," the graph indexing system materializes an ordered list of $\langle user.id,\ media.id,\ list(trusted\_user.id)\rangle$ tuples. Checking whether a write like W_3 (which expands the `TRUSTS` edge to include "music" between Bob and Alice) is relevant or locally applied to an index server requires the list of `MEDIA` nodes that Alice `ENJOYS`. This list could easily bloat the Ticket should we take this approach.

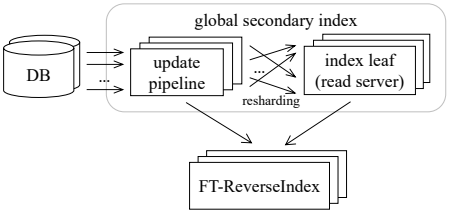

Figure 8: Stages of the asynchronous index update pipeline inform FT-RI as they process writes.

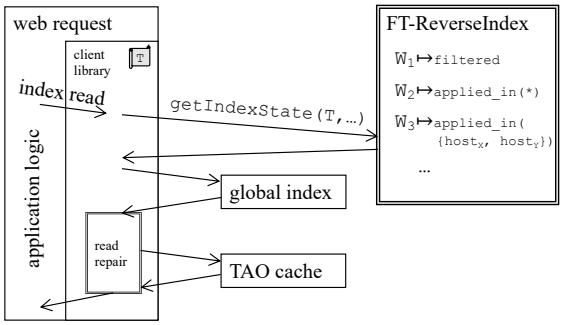

Figure 9: Web request flow with FT-ReverseIndex.

Another option we rejected was to ignore relevance checking for indexes and focus only on inclusion. This would require plumbing information about the replication water marks of all shards through the index update pipeline, perhaps using a compressed vector clock scheme like Occult [41] to avoid the need to deterministically merge across millions of replication streams. To distinguish lack of new updates from staleness in replication, each stream needs heartbeats, which results in a lot of overhead for cold shards and small indexes.

Our solution is to build an inverted index from writes to the actions taken by the index update pipeline. We store this in a stateful component named *FlightTracker-ReverseIndex* (FT-RI). We describe the interactions between the index update pipeline and FT-RI shown in Figure 8 with the example of W_3 and the intersection index. Based on W_3's type and index schemas, FT-RI determines the indexes W_3 could affect and initially assumes it could affect every row in those indexes. As W_3 goes through the update pipeline, each stage of the pipeline informs FT-RI when it is about to filter out W_3 for some or

all indexes. We also require the update pipeline to propagate W_3's metadata all the way to the index leaf servers unless W_3 is filtered out entirely. The update pipeline determines that W_3 matters only for index rows of the form $\langle 17,\ media.id,\ 42\rangle$ in the intersection index and informs FT-RI. When a leaf applies the index row updates generated due to W_3, it informs FT-RI of its server identifier and the part of the index state was updated. This way, FT-RI narrows down the scope of the indexes and read queries W_3 might affect.

As shown in Figure 9, to perform both relevance filtering and inclusion checking for a Ticket-inclusive index read, the client library first sends the query and the Ticket to FT-RI. FT-RI then returns the subset of writes that might be relevant (since they have not been reported as filtered) and not yet included (since they are still missing from the index leaves).

The client consults FT-RI before the query is sent to the index, so the set of missing writes may include false positives. False negatives would lead to RYW violations, so they must be avoided. To minimize the false positive rate without introducing any false negatives, FT-RI returns a map from writes to the physical servers where it may be missing. The client checks this information against the query execution plan, in case the stale server was not actually consulted. It can also be used to make intelligent replica choices and to retry only the stale portion of a query.

FT-RI accumulates a set of irrefutable facts about writes, so its internal state is a CRDT. It exploits the same single-round quorum protocol as FlightTracker (§ 5) for replication. FT-RI also shares much of the same infrastructure and is deployed as a RAM-only regional service.

6.4 Strategies to handle local staleness

This section describes ways to get the correct result when a potentially relevant write might be missing from the local data store. Our approaches fall into two categories: when the Ticket enumerates individual writes, the data store can request the data from upstream and cache the result for the next reader; if the Ticket contains a contiguous prefix, such as after compaction, we generally only reevaluate the query (on a different replica or at a later time), as it is expensive or impossible to request the contiguous prefix. In production, we use every strategy below but index repair.

Delay and retry: When we realize a data store is stale, a simple option is to just try again later. This strategy is not sufficient on its own, but it can be used as a first try to reduce the frequency of a more costly strategy.

Replica selection: Data stores are replicated for read availability. When one replica is stale, we can contact another replica that is up-to-date, especially if it is nearby. This strategy can lead to correlated failures such as thundering herd, so we only use it for low-volume workloads or behind a cache.

Consistency miss: When a Ticket identifies individual writes by key, caches that keep per-key versions (§ 4.1) can easily determine which data items are stale. They can use their normal miss-handling logic to pull data about the missing writes from their upstream source, passing on the Ticket to recursively ensure visibility.

Even client-side hot object caches can similarly take consistency misses, which otherwise rely on TTLs to get fresh data. This is integral to our tolerance of read hot spots (§ 6.5).

Index bypass (re-materialization): Indexing systems have the option to fall back to the source of normalized data to answer a read query, though this is an expensive option. Quite a few of our indexes are materialized on-demand, so this fallback functionality is already regularly exercised.

Read repair: Read repair looks for possible matches to components of an index predicate among the writes in a Ticket, uses point queries to a non-index store like TAO to evaluate the full predicate, and then fixes the index result accordingly. Read repair can reduce complexity and latency. Consider the example in § 3.4 where we want to find Bob's trusted users who also enjoy Song 55. If W_3 on edge $\langle 17, \texttt{TRUSTS}, 42\rangle$ is in the Ticket and the read repair library sees that node 42 (Alice) `ENJOYS` Song 55 from TAO, it adds node 42 into the result set.

Index read repair does not completely avoid extra communication on following queries, like consistency misses in a cache, but it avoids the need for future cross-region calls.

FT-RI filters out writes we do not need to repair. As shown in Figure 9, the client library queries FT-RI to find which of Bob's relevant writes have not yet been applied before querying the index and read repair. We initially tested read repair without FT-RI, treating every write a user had made in the last 60 seconds as undetermined. FT-RI for list intersection indexes has only a modest effect on the average number of edges to be checked for read repair, but it provides a dramatic reduction for the worst cases.

Read repair has its limitations. Firstly, certain indexes with aggregation cannot be read repaired. For example, if an index query only returns the size of the intersection, the read repair library would not know which writes have been applied in the result. Fortunately, the vast majority of our online index usage returns set results that can be repaired. Secondly, client-side read repair for complex indexes, such as ones that require traversing 3+ hops in the graph or those with large fan-outs, could duplicate the transform and processing logic of the update pipeline and index leaves, resulting in extra complexity. The above challenges are akin to those encountered in deferred incremental view maintenance in the database community [23, 24, 47, 58].

Index repair: Repairing the index by synchronously invoking the index update logic is more complex, but avoids many of the limitations of client-side read repair. We have not yet explored this option.

6.5 Handling hot spots

Handling linchpin objects is one of the major challenges of a social networking workload [10]. Read hot spots are a much

bigger concern than write hot spots in our read-heavy workload. Caches like TAO handle read hot spots by storing more local copies of the data, including on the client-side. Ticket-inclusive reads for cache queries are cache-able, preserving this hot spot tolerance. We cannot always cache post-repair index results, but the data fetched to perform repair is always locally cacheable.

Aside from hot objects for the overall system, FlightTracker has its own hot spots: since FlightTracker is sharded by `session_id`, it has different hot spot patterns from the underlying data stores. These hot spots are more likely due to user-triggered actions such as batch processing or from custom sessions (§ 7.3). To alleviate write hot spots, FlightTracker's client library batches metadata writes without concerns for sacrificing write availability, since all FlightTracker writes are conflict-free. On the FlightTracker server side, we proactively detect sessions that are frequently accessed. For a hot session that spawns many web requests and thus results in many metadata reads, we coalesce these metadata reads into short time buckets, and respond to all reads in a bucket with the same response. These strategies have eliminated hot spots as a significant error source for FlightTracker (§ 8).

7 Beyond RYW: Explicit write visibility

Certain applications need visibility guarantees beyond a single end user or across regions, where our default user-centric RYW consistency falls short. To obtain desired visibility guarantees, we enable them to explicitly manipulate Tickets or customize FlightTracker session IDs. These applications are responsible for explicitly identifying the writes and preventing the Tickets from growing too large.

7.1 Embedding Tickets in notifications

Facebook's notification infrastructure for GraphQL Subscriptions [48] fans out to all subscribers when a publisher event occurs. To render personalized notifications for each subscriber, GraphQL queries TAO in the subscriber regions. This pub-sub system races with TAO replication. To ensure that the query sees all of the writes associated with the event, we include and pass along the original publisher's Ticket. When rendering the notification, GraphQL transiently joins it with the subscriber's Ticket to query TAO. This is all hidden in the product infrastructure layer from product developers.

Subscriptions with a high subscribers fanout could result in a storm of consistency misses. Though TAO would only go cross-region once for this data, many requests would be stalled waiting for the result. Thus, we prefetch data in the Ticket into the local region's TAO before notification fanout.

7.2 Data-derived additional sessions

When a user performs a write that includes another person's User ID, such as when Bob created a `TRUSTS` edge to Alice, the write is naturally associated with the other user's FlightTracker session. For some edge types, we act on this by having the client library perform extra `appendWrites` calls, pushing the write to both the normal RYW session and the session identified by the destination node. These additional writes are sent to every region. We do not push the entire writing user's Ticket into the data-derived additional sessions; only the user-terminated edge write gets strengthened visibility guarantees. As users are highly connected [7], this conservative choice avoids the potential for super-linear growth of write sets.

7.3 Explicit global sessions

Some applications need visibility guarantees beyond a single end user or across region. We allow them to customize their session IDs and configure quorum and compaction in FlightTracker on a per-use-case basis.

Facebook's async job scheduling framework, similar to Celery or Resque [1, 5], enables web requests to schedule followup jobs such as sending email invites or long running migrations. These jobs may run in any region, but all of the writes from the original user session must be visible. To provide this guarantee, we use `job_id` as the session ID, which is the same for all tasks that are part of a job. Given the relative read-write ratio, we require a write to be replicated to most replicas in all regions and a read to be read from a few (usually region-local) replicas. We provide a utility function for the job framework to collect the writes the web request has done and send to FlightTracker under the appropriate job ID. When a job starts, it fetches a Ticket from FlightTracker using its job ID and uses Ticket-inclusive reads thereafter.

We also use global sessions for some TAO objects as an alternative to TAO's critical reads. Critical reads ensure write visibility by proxying reads to the region of the object's database primary, at the expense of increased latency, reduced efficiency, and reduced availability. If we record all writes to this object in a global session using its ID, we can replace the critical read: querying region-local FlightTracker to get a Ticket for this session and querying that object with a Ticket-inclusive read will return the latest successful (or newer) write. This approach shifts the cross-region latency to write time and increases read availability.

8 Evaluation

FlightTracker allows Facebook to get the read efficiency, hot spot tolerance, and high availability of eventual consistency while providing RYW consistency with a rich notion of user sessions that spans many stateful services.

8.1 Environment

Facebook serves millions of user requests per second. These user requests amplify to more than ten billion read queries per second to our online graph data stores, which also process tens of millions of writes per second.

Each of our stateful data stores, such as TAO and its indexes,

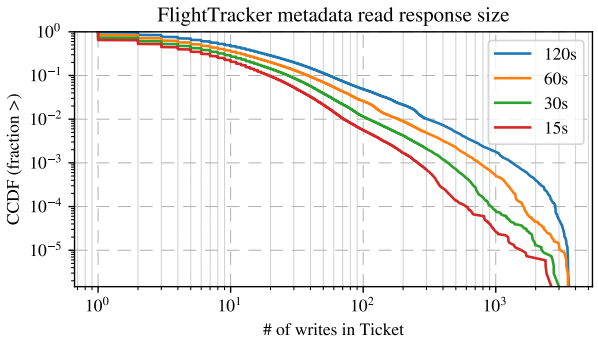

Figure 10: The size distribution of FlightTracker read responses in our production environment for different global compaction thresholds.

are deployed across over ten datacenter regions. Cache and indexing systems maintain asynchronously updated replicas in every region, while database replicas are present only in some regions. More than 99% of cache and index queries are served without any cross-region communication.

8.2 FlightTracker operational characteristics

FlightTracker has been in production for over four years. It manages RYW consistency for two database technologies, two cache types (including TAO), and three indexing systems. FlightTracker serves more than 100 million Ticket reads and 20 million Ticket writes per second.

We measured FlightTracker's availability as observed by the client over 30 days. Measuring errors from the client side offers an end-to-end picture, because it includes unavailability due to misconfiguration, networking issues, and collateral damage from other problems. FlightTracker's overall read error rate was 1.1×10^{-7}. When examining the availability data for 15-minute buckets, all but 8 buckets over the month of data indicated at least 99.9999% of read availability. FlightTracker's write availability was an order of magnitude higher than the write availability of the underlying databases.

8.3 FlightTracker overheads

Request and response sizes: The bane of explicit write-tracking techniques is handling large write sets. The Tickets fetched from FlightTracker contain all recent writes for a user that are not globally compacted (§ 4.2), so they tend to be the largest explicit write sets passed around in our systems. Figure 10 shows the size distribution of metadata read responses in production for different global compaction thresholds, measured in the number of writes in the returned Ticket. In production, we use 60s as the default, but as shown, extending it to 2 minutes does not significantly bend the curve.

Ticket serialization includes compression using LZ4 [25]. Figure 11 shows that this provides a useful benefit for Tickets with more individual writes, improving encoding efficiency by up to a factor of three. Table 1 shows that cropping in the client is effective; Ticket sizes attached to read queries are much smaller than the full write set pulled from FlightTracker.

Operation	Avg	P50	P99
FlightTracker metadata read response	250	0	2805
FlightTracker metadata write request	156	129	447
Ticket-inclusive read from cache	110	0	450
Ticket-inclusive read from indexes	225	152	607

Table 1: Serialized sizes of Tickets attached on various requests and responses, in bytes.

Operation	Avg	P50	P99
FlightTracker read	288 µs	226 µs	1.4 ms
FlightTracker write	376 µs	326 µs	1.5 ms
FT-ReverseIndex read	304 µs	236 µs	1.5 ms
FT-ReverseIndex update	428 µs	311 µs	1.2 ms

Table 2: Client-measured latency of FlightTracker and FT-RI.

Service	CPU	RAM
Application (web) servers	0.7%	0.06%
TAO L1 and L2 cache	0.8%	0.01%
Indexes and materialized views	0.98%	2.6%

Table 3: Relative CPU and memory costs of all code paths related to Ticket, FlightTracker, or FT-ReverseIndex.

Latency: FlightTracker and FT-RI have low latency for both reads and writes, as shown in Table 2. Both of these services are RAM-only and process all of their reads and writes from the local datacenter region. Queries for custom use cases (§ 7.3) are excluded in the table.

Footprint: The footprint of FlightTracker includes extra work and data in client libraries, extra work and space inside the data stores to enable Ticket-inclusive reads, and servers devoted exclusively to running the FlightTracker and FT-RI services. Table 3 shows that FlightTracker-related code paths consume only a small amount of the CPU and memory in clients and Ticket-enabled query-serving systems. The FlightTracker and FT-RI services use less than 2% as many servers as TAO and its indexes.

Extensibility: The Ticket abstraction is designed to be extended to handle new databases and new ways of encoding write metadata for the benefit of new projections. Since it was deployed to production, we have changed the Ticket Thrift schema 22 times, and we have made changes to the core Ticket join logic 50 times. Extending the Ticket abstraction to cover a new database does not increase the serialized size, but it does increase the heap footprint of the deserialized C++ objects. FlightTracker's RAM consumption increased by 5% when we added support for a second type of database.

8.4 FlightTracker effectiveness

RYW for caches: FlightTracker enabled our caches to no longer rely on fixed communication topology to provide RYW consistency. It provided an opportunity to apply additional techniques to improve efficiency and reliability for our caches.

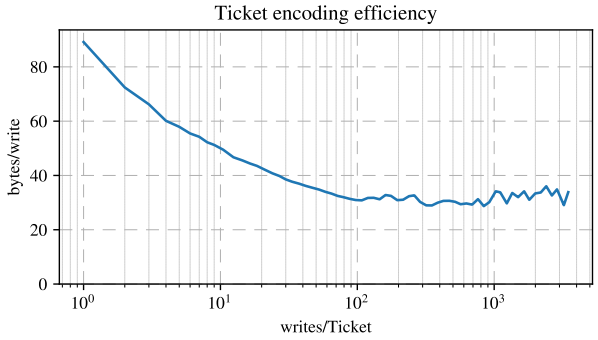

Figure 11: LZ4 improves serialized encoding efficiency by up to a factor of three for Tickets with more individual writes.

Today, 0.2% of the TAO reads have a non-empty Ticket attached, and 3% of those reads are for updates that have not yet been replicated via the per-shard replication stream. We allocate around 40MB per TAO instance for caching the result of Ticket-inclusive reads, resulting in a hit rate between 30% and 80% depending on the type of query. This hit rate is not evenly distributed: frequently read hot objects account for the bulk of hits. Fewer than 3% of TAO reads that end up going across regions are due to consistency misses.

Ticket-inclusive reads reduce cross-region traffic when they replace primary-DB-only queries for use cases that need stronger consistency guarantees. In one extreme use case where the cache only tracked per-shard replication progress, Ticket-inclusive reads reduced the percentage of queries going upstream from 20-40% to near 0%.

RYW for indexes: FlightTracker identifies that between 0.01% to 0.4% of indexing reads can benefit from read repair or other staleness handling strategies.

Explicit use cases: In the event delivery use case (§ 7.1), 3% of the subscribers' reads are for publishers' recent writes. 0.5% of these reads would have returned stale data without the publisher's Ticket. Six use cases benefit from global sessions (§ 7.3), totaling 46k writes and 700k reads per second. They often set large write quorums to optimize for read availability and latency.

8.5 Experience and lessons learned

An early lesson was that identifying the appropriate user for a web request was much more difficult than we originally expected. Request endpoints may be invoked before login or after logout; internal applications may track user contexts using bespoke mechanisms; and applications may involve multiple identities, such as when a user manages a business account. Getting high query coverage involved a lot of manual work to discover alternate user contexts and identify endpoints that are not expected to be associated with a user.

Global sessions tend to be used for metadata stored in TAO, such as for product flows modeled as state machines. The addition of global sessions to a code base is often done fairly late in the product development cycle, to fix issues neglected in the initial design. Our ability to strengthen write visibility for such call sites, without data migrations or schema changes, is an important part of making RYW a reasonable default.

The applications that cause the most challenge operationally need RYW consistency the least. These tend to be internal applications that perform batch processing or involve massive fan-out. They often cause write hot spots in the data stores but rarely read what they wrote afterwards.

Closing consistency loopholes with FlightTracker revealed the underlying systems were not actually eventually consistent. We have found low-probability bugs that cause permanent inconsistencies in TAO, graph indexes, and even database replication. These bugs were previously difficult to notice, as they were outnumbered by transient inconsistencies. Ticket-inclusive reads should never return old data, so now that we have FlightTracker even a single occurrence of a stale result is actionable. Bugs leading to permanent inconsistencies included protocol flaws, incorrect handling of error conditions, and relying on data invariants that were not honored by all historical data.

9 Limitations and future work

Our approach still relies on region-sticky user routing. We could avoid this limitation by always using global quorums like in § 7.3, but this would increase latency. We plan to eventually make user RYW sessions global by maintaining a map from user to region in FlightTracker, rehoming sessions when the mapping changes.

The relative efficiency of our solution depends on amortizing the cost of the metadata reads across many TAO queries, and depends on the set of writes being relatively small. Environments with fewer reads have a different set of tradeoffs. This limitation is less applicable to index queries, because those tend to do more work per operation.

FlightTracker does not provide consistency for "unacknowledged successes." As described previously in § 5.3, unacknowledged successes happen when a data write has a client error like a timeout or if the metadata write fails. We have not seen this to be a problem in practice, probably because the issue exists even without FlightTracker.

Some queries are difficult to repair: TAO top-N queries for large edge lists result in unnecessary consistency misses; index queries to a materialized aggregation (such as counts) can be detected as stale by FT-RI, but the stale result cannot be fixed with read repair; and list intersection queries that involve more than two lists are also difficult to repair.

The FlightTracker service compacts Tickets into a timestamp bound, so that we will take a consistency miss if replication exceeds the global compaction bound of 60s (§ 4.2). The global compaction bound is not fully rolled out as of publication time, so tail latency events in the replication pipeline can result in RYW violations. This has not been a big issue in practice because it requires that the first read of a write occurs after the global compaction interval but before replication.

Although some of our motivations are specific to Facebook's workload, our desire to provide user-centric sessions is widely shared [42,55], as is our desire to extend consistency guarantees to global indexes [3,12,30,34]. Cache invalidation is also a perennial challenge for systems at all scales.

Our FlightTracker approach is generalizable: designed for heterogeneous data stores, Tickets can easily be extended to other data stores without much overhead (§ 4.1); the API extensions data stores need to implement (Figure 3) do not require core replication protocol changes and have relatively small overhead (§ 8.3); the client library where a lot of the FlightTracker logic lives can be implemented and rolled out gradually. Our approach is especially beneficial when trying to retrofit indexing systems, because it allows us to separate the reverse metadata index into its own component.

10 Related Work

Stronger consistency atop eventually consistent stores: Many eventually consistent systems offer options to opt for stronger consistency levels. Systems such as Cassandra [37], Riak [6], and RedBlue [39] provide strong consistency either by routing read requests to the leader or by adjusting their commit protocols. To provide bounded staleness, Azure CosmosDB [21] could backpressure writes. In contrast, FlightTracker serves most reads from a single local replica.

Index consistency: Most of these systems, including Amazon's DynamoDB [11, 12] and Google AppEngine Datastore [31], do not extend the stronger consistency levels to global secondary indexes. Twitter's Manhattan [49] extends RYW to global secondary indexes by including them in a cross-shard transactional write, doubling latency [34, 55]. Couchbase [3] supports RYW for reads to its global indexes using a timestamp in the client session. It accomplishes this by deterministically merging updates to all shards, limiting scalability in the number of shards.

Bailis et al. [15] proved that index consistency can be implemented with better availability characteristics than approaches that include indexes in general-purpose transactions.

Implementing RYW sessions: Session RYW [51] is intuitive and implementable with low overhead [14, 27]. Bayou [28] and Pileus [52] provide session guarantees that span multiple servers by managing the session state in their client libraries. Bermbach et al. [19] similarly observed that client-centric consistency should focus on end users; their approach nonetheless assumes that a session is sticky to a single application server. PathStore [42] address the same challenge where clients interact with multiple data store replicas by using a session migration protocol on *every* replica switch. In contrast, FlightTracker manages session state in an intermediate layer between the client and the data servers.

Write-set tracking for stronger consistency: Systems like COPS [40] and SwiftCloud [56] track dependent write sets to provide causal consistency. They also provide client contexts that are similar to FlightTracker sessions. BoltOn [16] layers causal consistency guarantees via a shim over eventually consistent stores. Its design shares similar principles as FlightTracker, aiming to retain the desirable properties of eventually consistent stores. For these systems, the dependency sets need to be stored in the database and cached on the client-side.

TxCache [46] provides transactional (but possibly stale) consistency for application-level caching and uses the terms *staleness miss* and *consistency miss* (both of which are included in our use of the term consistency miss). Its design focuses on single datacenter and treats materialized views as cacheable results from user-specified functions, which is insufficient for our applications.

To reduce the metadata size, systems like Occult [41] and Wukong+S [57] use structural or temporal properties to compress write sets and vector timestamps. FlightTracker uses CRDT inflation to compact Tickets and trims irrelevant writes to reduce network overhead, but mainly avoids metadata size explosion by providing a weaker consistency level.

Tradeoff between cache hit rate and consistency: Zanzibar [44] is built on top of Google's linearizable Spanner [26], but chooses to expose a weaker consistency model to clients to improve its read efficiency and latency. Its zookies play a similar role to FlightTracker Tickets, encapsulating consistency information, but they are used only by Zanzibar itself.

CRDT quorum protocols: The single-round FlightTracker protocol is at its core CRDT [50] using quorum replication [29]. Gryff [22] and CURP [45] similarly leverage commutativity of writes. Because FlightTracker does not need atomicity, a single round suffices.

11 Conclusion

This paper introduces FlightTracker, our approach for providing RYW consistency for Facebook's social graph. FlightTracker operates in a read-optimized ecosystem of asynchronously replicated caches, database replicas, and indexes. It preserves the read efficiency, hot spot tolerance, and loose coupling benefits of eventual consistency, and it has allowed us to circumvent the scaling challenges we encountered when using write-through caching for consistency.

Acknowledgements

We thank the following people for critical contributions to the systems in this paper: Tina Park, Kevin Ventullo, Christopher Small, Andrew Bass, Tushar Pankaj, David Goode, Soham Shah, Lu Pan, Gordon (Zhuo) Huang, Brendan Forsyth, Neil Wheaton, Shilpa Lawande, and Tony Savor.

We thank our shepherd Malte Schwarzkopf and our reviewers for raising the bar on feedback quality. We thank Wyatt Lloyd, Mahesh Balakrishnan, and Rob Lyerly for their many valuable suggestions on the paper.

References

[1] Celery: Distributed Task Queue. http://www.celeryproject.org/.

[2] CockroachDB. https://www.cockroachlabs.com/.

[3] Couchbase Global Secondary Indexes. https://docs.couchbase.com/server/6.5/learn/services-and-indexes/indexes/global-secondary-indexes.html.

[4] LogDevice. https://logdevice.io/.

[5] Resque. http://resque.github.io/.

[6] Riak. https://riak.com/products/riak-kv/.

[7] Three and a Half Degrees of Separation. https://research.fb.com/blog/2016/02/three-and-a-half-degrees-of-separation/.

[8] Thrift. http://thrift.apache.org/.

[9] AHAMAD, M., NEIGER, G., BURNS, J. E., KOHLI, P., AND HUTTO, P. W. Causal Memory: Definitions, Implementation, and Programming. *Distributed Computing 9*, 1 (1995), 37–49.

[10] AJOUX, P., BRONSON, N., KUMAR, S., LLOYD, W., AND VEERARAGHAVAN, K. Challenges to Adopting Stronger Consistency at Scale. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems* (USA, 2015), HOTOS'15, USENIX Association, p. 13.

[11] AMAZON. Design Patterns Using Amazon DynamoDB. https://www.slideshare.net/AmazonWebServices/design-patterns-using-amazon-dynamodb.

[12] AMAZON. Improving Data Access with Secondary Indexes. https://docs.amazonaws.cn/en_us/amazondynamodb/latest/developerguide/SecondaryIndexes.html.

[13] ANTONOPOULOS, P., BUDOVSKI, A., DIACONU, C., HERNANDEZ SAENZ, A., HU, J., KODAVALLA, H., KOSSMANN, D., LINGAM, S., MINHAS, U. F., PRAKASH, N., PUROHIT, V., QU, H., RAVELLA, C. S., REISTETER, K., SHROTRI, S., TANG, D., AND WAKADE, V. Socrates: The New SQL Server in the Cloud. In *Proceedings of the 2019 International Conference on Management of Data* (New York, NY, USA, 2019), SIGMOD '19, Association for Computing Machinery, p. 1743–1756.

[14] BAILIS, P., DAVIDSON, A., FEKETE, A., GHODSI, A., HELLERSTEIN, J. M., AND STOICA, I. Highly Available Transactions: Virtues and Limitations. *Proc. VLDB Endow. 7*, 3 (Nov. 2013), 181–192.

[15] BAILIS, P., FEKETE, A., FRANKLIN, M. J., GHODSI, A., HELLERSTEIN, J. M., AND STOICA, I. Coordination Avoidance in Database Systems. *Proc. VLDB Endow. 8*, 3 (Nov. 2014), 185–196.

[16] BAILIS, P., GHODSI, A., HELLERSTEIN, J. M., AND STOICA, I. Bolt-on Causal Consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2013), SIGMOD '13, Association for Computing Machinery, p. 761–772.

[17] BAILIS, P., VENKATARAMAN, S., FRANKLIN, M. J., HELLERSTEIN, J. M., AND STOICA, I. Probabilistically Bounded Staleness for Practical Partial Quorums. *Proc. VLDB Endow. 5*, 8 (Apr. 2012), 776–787.

[18] BAQUERO, C., ALMEIDA, P. S., CUNHA, A., AND FERREIRA, C. Composition of State-based CRDTs. *HASLab, May* (2015).

[19] BERMBACH, D., KUHLENKAMP, J., DERRE, B., KLEMS, M., AND TAI, S. A Middleware Guaranteeing Client-Centric Consistency on Top of Eventually Consistent Datastores. In *2013 IEEE International Conference on Cloud Engineering (IC2E)* (2013), IEEE, pp. 114–123.

[20] BRONSON, N., AMSDEN, Z., CABRERA, G., CHAKKA, P., DIMOV, P., DING, H., FERRIS, J., GIARDULLO, A., KULKARNI, S., LI, H., AND ET AL. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference* (USA, 2013), USENIX ATC'13, USENIX Association, p. 49–60.

[21] BROWN, M. Consistency Levels in Azure Cosmos DB. https://docs.microsoft.com/en-us/azure/cosmos-db/consistency-levels.

[22] BURKE, M., CHENG, A., AND LLOYD, W. Gryff: Unifying Consensus and Shared Registers. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)* (2020), pp. 591–617.

[23] CHO, S., AVERBUKH, R., ZHANG, Y., CARTER, A., AND JAN, J. A. Partial Update: Efficient Materialized View Maintenance in a Distributed Graph Database. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)* (2018), pp. 1477–1488.

[24] COLBY, L. S., GRIFFIN, T., LIBKIN, L., MUMICK, I. S., AND TRICKEY, H. Algorithms for Deferred View Maintenance. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 1996), SIGMOD '96, Association for Computing Machinery, p. 469–480.

[25] COLLET, Y. LZ4 – Extremely Fast Compression. https://github.com/lz4/lz4.

[26] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J. J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., HSIEH, W., KANTHAK, S., KOGAN, E., LI, H., LLOYD, A., MELNIK, S., MWAURA, D., NAGLE, D., QUINLAN, S., RAO, R., ROLIG, L., SAITO, Y., SZYMANIAK, M., TAYLOR, C., WANG, R., AND WOODFORD, D. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst. 31*, 3 (Aug. 2013).

[27] CROOKS, N., PU, Y., ALVISI, L., AND CLEMENT, A. Seeing is Believing: A Unified Model for Consistency and Isolation via States. *CoRR abs/1609.06670* (2016).

[28] EDWARDS, W. K., MYNATT, E. D., PETERSEN, K., SPREITZER, M. J., TERRY, D. B., AND THEIMER, M. M. Designing and Implementing Asynchronous Collaborative Applications with Bayou. In *Proceedings of the 10th Annual ACM Symposium on User Interface Software and Technology* (New York, NY, USA, 1997), UIST '97, Association for Computing Machinery, p. 119–128.

[29] GIFFORD, D. K. Weighted Voting for Replicated Data. In *Proceedings of the Seventh ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1979), SOSP '79, Association for Computing Machinery, p. 150–162.

[30] GJENGSET, J., SCHWARZKOPF, M., BEHRENS, J., ARAÚJO, L. T., EK, M., KOHLER, E., KAASHOEK, M. F., AND MORRIS, R. Noria: Dynamic, Partially-Stateful Data-Flow for High-Performance Web Applications. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (USA, 2018), OSDI'18, USENIX Association, p. 213–231.

[31] GOOGLE. Data consistency in Datastore queries. https://cloud.google.com/appengine/docs/standard/java/datastore/data-consistency#query-data-consistency.

[32] HERLIHY, M. P., AND WING, J. M. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst. 12*, 3 (July 1990), 463–492.

[33] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX annual technical conference* (2010), vol. 8, p. 9.

[34] KATOORU, K. Native Secondary Indexing in Manhattan. https://blog.twitter.com/engineering/en_us/topics/infrastructure/2018/native-secondary-indexing-in-manhattan.html.

[35] KREPS, J., NARKHEDE, N., RAO, J., ET AL. Kafka: A Distributed Messaging System for Log Processing.

[36] KULKARNI, S. S., DEMIRBAS, M., MADAPPA, D., AVVA, B., AND LEONE, M. Logical Physical Clocks. In *International Conference on Principles of Distributed Systems* (2014), Springer, pp. 17–32.

[37] LAKSHMAN, A., AND MALIK, P. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev. 44*, 2 (Apr. 2010), 35–40.

[38] LAMPORT, L. Paxos Made Simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (December 2001), 51–58.

[39] LI, C., PORTO, D., CLEMENT, A., GEHRKE, J., PREGUIÇA, N., AND RODRIGUES, R. Making Geo-Replicated Systems Fast as Possible, Consistent When Necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (USA, 2012), OSDI'12, USENIX Association, p. 265–278.

[40] LLOYD, W., FREEDMAN, M. J., KAMINSKY, M., AND ANDERSEN, D. G. Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, Association for Computing Machinery, p. 401–416.

[41] MEHDI, S. A., LITTLEY, C., CROOKS, N., ALVISI, L., BRONSON, N., AND LLOYD, W. I Can't Believe It's Not Causal! Scalable Causal Consistency with No Slowdown Cascades. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation* (USA, 2017), NSDI'17, USENIX Association, p. 453–468.

[42] MORTAZAVI, S. H., BALASUBRAMANIAN, B., DE LARA, E., AND NARAYANAN, S. P. Toward Session Consistency for the Edge. In *USENIX Workshop on Hot Topics in Edge Computing (HotEdge 18)* (2018).

[43] ONGARO, D., AND OUSTERHOUT, J. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual*

Technical Conference (USA, 2014), USENIX ATC'14, USENIX Association, p. 305–320.

[44] PANG, R., CACERES, R., BURROWS, M., CHEN, Z., DAVE, P., GERMER, N., GOLYNSKI, A., GRANEY, K., KANG, N., KISSNER, L., KORN, J. L., PARMAR, A., RICHARDS, C. D., AND WANG, M. Zanzibar: Google's Consistent, Global Authorization System. In *2019 USENIX Annual Technical Conference (USENIX ATC '19)* (Renton, WA, 2019).

[45] PARK, S. J., AND OUSTERHOUT, J. Exploiting Commutativity for Practical Fast Replication. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation* (USA, 2019), NSDI'19, USENIX Association, p. 47–64.

[46] PORTS, D. R. K., CLEMENTS, A. T., ZHANG, I., MADDEN, S., AND LISKOV, B. Transactional Consistency and Automatic Management in an Application Data Cache. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (USA, 2010), OSDI'10, USENIX Association, p. 279–292.

[47] SALEM, K., BEYER, K., LINDSAY, B., AND COCHRANE, R. How to Roll a Join: Asynchronous Incremental View Maintenance. *SIGMOD Rec. 29*, 2 (May 2000), 129–140.

[48] SCHAFER, D., AND KUENZEL, L. Subscriptions in GraphQL and Relay. https://graphql.org/blog/subscriptions-in-graphql-and-relay/.

[49] SCHULLER, P. 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.

[50] SHAPIRO, M., PREGUIÇA, N., BAQUERO, C., AND ZAWIRSKI, M. Conflict-free Replicated Data Types. In *Symposium on Self-Stabilizing Systems* (2011), Springer, pp. 386–400.

[51] TERRY, D. B., DEMERS, A. J., PETERSEN, K., SPREITZER, M. J., THEIMER, M. M., AND WELCH, B. B. Session Guarantees for Weakly Consistent Replicated Data. In *Proceedings of the Third International Conference on on Parallel and Distributed Information Systems* (Washington, DC, USA, 1994), PDIS '94, IEEE Computer Society Press, p. 140–150.

[52] TERRY, D. B., PRABHAKARAN, V., KOTLA, R., BALAKRISHNAN, M., AGUILERA, M. K., AND ABU-LIBDEH, H. Consistency-Based Service Level Agreements for Cloud Storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, Association for Computing Machinery, p. 309–324.

[53] THOMSON, A., DIAMOND, T., WENG, S.-C., REN, K., SHAO, P., AND ABADI, D. J. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2012), SIGMOD '12, Association for Computing Machinery, p. 1–12.

[54] VOGELS, W. Eventually Consistent. https://queue.acm.org/detail.cfm?id=1466448.

[55] YARMULA, A. Strong consistency in Manhattan. https://blog.twitter.com/engineering/en_us/a/2016/strong-consistency-in-manhattan.html.

[56] ZAWIRSKI, M., PREGUIÇA, N., DUARTE, S., BIENIUSA, A., BALEGAS, V., AND SHAPIRO, M. Write Fast, Read in the Past: Causal Consistency for Client-Side Applications. In *Proceedings of the 16th Annual Middleware Conference* (New York, NY, USA, 2015), Middleware '15, Association for Computing Machinery, p. 75–87.

[57] ZHANG, Y., CHEN, R., AND CHEN, H. Sub-Millisecond Stateful Stream Querying over Fast-Evolving Linked Data. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, Association for Computing Machinery, p. 614–630.

[58] ZHUGE, Y., GARCÍA-MOLINA, H., HAMMER, J., AND WIDOM, J. View Maintenance in a Warehousing Environment. *SIGMOD Rec. 24*, 2 (May 1995), 316–327.