Existential Consistency: Measuring and Understanding Consistency at Facebook

Haonan Lu*†, Kaushik Veeraraghavan†, Philippe Ajoux†, Jim Hunt†, Yee Jiun Song†, Wendy Tobagus†, Sanjeev Kumar†, Wyatt Lloyd*†

*University of Southern California, †Facebook, Inc.

Abstract

Replicated storage for large Web services faces a trade-off between stronger forms of consistency and higher performance properties. Stronger consistency prevents anomalies, i.e., unexpected behavior visible to users, and reduces programming complexity. There is much recent work on improving the performance properties of systems with stronger consistency, yet the flip-side of this trade-off remains elusively hard to quantify. To the best of our knowledge, no prior work does so for a large, production Web service.

We use measurement and analysis of requests to Facebook's TAO system to quantify how often anomalies happen in practice, i.e., when results returned by eventually consistent TAO differ from what is allowed by stronger consistency models. For instance, our analysis shows that 0.0004% of reads to vertices would return different results in a linearizable system. This in turn gives insight into the benefits of stronger consistency; 0.0004% of reads are potential anomalies that a linearizable system would prevent. We directly study local consistency models—i.e., those we can analyze using requests to a sample of objects—and use the relationships between models to infer bounds on the others.

We also describe a practical consistency monitoring system that tracks ϕ-consistency, a new consistency metric ideally suited for health monitoring. In addition, we give insight into the increased programming complexity of weaker consistency by discussing bugs our monitoring uncovered, and anti-patterns we teach developers to avoid.

SOSP'15, October 4–7, 2015, Monterey, CA.
ACM 978-1-4503-3834-9/15/10.
http://dx.doi.org/10.1145/2815400.2815426

1. Introduction

Replicated storage is an important component of large Web services and the consistency model it provides determines the guarantees for operations upon it. The guarantees range from eventual consistency, which ensures replicas eventually agree on the value of data items after receiving the same set of updates to strict serializability [12] that ensures transactional isolation and external consistency [25]. Stronger consistency guarantees often require heavier-weight implementations that increase latency and/or decrease throughput [5, 13, 26, 37]. As a result, many production systems [14, 15, 19, 24, 32, 42, 46] choose weaker forms of consistency in order to provide low latency and high throughput.

These weaker forms of consistency have two primary drawbacks. First, they admit executions with user-visible *anomalies*, i.e., strange behavior that defies user expectations. A common example is out-of-order comments on a social network post, e.g., Alice comments on a post, Bob comments on the same post after seeing Alice's comments, and then Charlie sees Bob's comment appear before Alice's. The second drawback of weaker consistency models is that they increase *programming complexity*, i.e., programmers working on systems with weaker models must reason about and handle all the complex cases. A common example is the loss of referential integrity, e.g., Alice uploads a photo, and then adds it to an album. In some weaker consistency models, the programmer must reason about an album with references to photos that do not yet exist.

There has been much recent work on providing intermediate [3, 7, 21, 22, 38, 39], and even strong consistency [6, 9, 17, 18, 23, 27, 31, 36, 39, 41, 43, 50–52, 54, 56] with increasingly high throughput and low latency. Yet the flip side of this trade-off remains elusively difficult to quantify; to the best of our knowledge there is no prior work that does so for a large, production Web service. Without an understanding of the consistency benefits of intermediate and strong consistency, it is difficult to fully evaluate how they compare to weaker models, and each other.

This work takes the first step towards quantifying those benefits by measuring and analyzing requests to the social graph at Facebook. We focus our study on a social network because it is the application that has motivated much of the recent research boom in replicated storage. We also focus on it because it provides a more interesting trade-off between performance and consistency than some other applications that require strong forms of consistency, e.g., the ads served by the F1 database [49], which motivates the strongly consistent Spanner system [17].

Facebook's replicated storage for its social graph is a combination of a two-level cache and a sharded, single-master-per-shard database. The caches are grouped into clusters. Within a cluster, per-object sequential and read-after-write consistency are provided. Across the entire system, eventual consistency is provided. We perform two types of analysis on this system: a principled analysis and a practical analysis. The principled analysis identifies when the results of the system differ from what is allowed by stronger consistency models, i.e., what anomalies occur in the eventually consistent production system. The practical analysis is used as a real-time monitoring tool. It is also a useful tool for finding bugs in code written on top of the storage system. These bugs give us insight into the types of mistakes that can happen due to the increased programmer complexity due to weaker consistency models.

We conduct the principled analysis by logging all the requests to a small random sample of the social graph, and by running offline consistency checkers on those logs. We have created checkers that identify when the real, eventually consistent system returns results that are disallowed by stronger consistency models. We have checkers for local consistency

models, e.g., linearizability [29], per-object sequential consistency [16], and read-after-write consistency, which can be accurately measured by a random sample. In addition, we use the theoretical relationships between consistency models to infer bounds on the results for non-local consistency models, e.g., causal consistency [1, 33], which cannot be accurately measured by a random sample. The results of these checkers directly measure or bound how often anomalies occur and give insight into the benefits of different consistency models in reducing the frequency of anomalies.

Running the principled analysis in real-time would be equivalent to implementing a system with stronger consistency guarantees, and running it in parallel with the eventually consistent system. To avoid that overhead, we instead use a practical online consistency checker for real-time health monitoring of the replicated storage system. The practical checker measures ϕ-consistency, a new metric that can be computed in real-time. It reflects the frequency of all replicas returning the same results for a user's read request. We define ϕ-consistency formally, relate it to principled consistency models, and explain how it is used to monitor the health of the system. The practical checker has been deployed at Facebook since 2012.

The contributions of this paper include:

- The first systematic analysis of the benefits of stronger consistency models in a large-scale production system.
- A principled approach for identifying anomalies, and a practical approach for real-time health monitoring of a weakly consistent system.
- Insights into the effects of increased programmer complexity due to weaker consistency models through a discussion of bugs our monitoring system has uncovered, and anti-patterns we teach developers to avoid.

We present background on Facebook's replicated storage and consistency models in Section 2. We present our principled analysis in Section 3 and our practical analysis in Section 4. We discuss experience in Section 5. We review related work in Section 6; and we conclude in Section 7.

2. Background

This section covers background on Facebook's replicated storage and consistency models.

2.1 Facebook's Replicated Storage

The replicated storage that holds the social graph at Facebook uses a graph data model stored in a two-level cache and a backing database.

Data Model Facebook models entities and the relationships between them with vertices and directed edges in a graph data model.[1] *Vertices* are typed objects identified by a 64-bit unique id and are stored with a type-specific dictionary. For instance, vertex 370 could be a post object with the content and creation time of the post stored in the dictionary, and vertex 450 could be a user object with the hometown and join date stored in the dictionary.

Directed edges in the graph are also typed objects stored with a type-specific dictionary, but they are identified by their endpoints. For instance, edge$(370, 450)$ could be a posted_by edge that indicates user 450 created post 370 with an empty dictionary. Edges are indexed

[1] Vertices and edges are also called *objects* and *associations* [14].

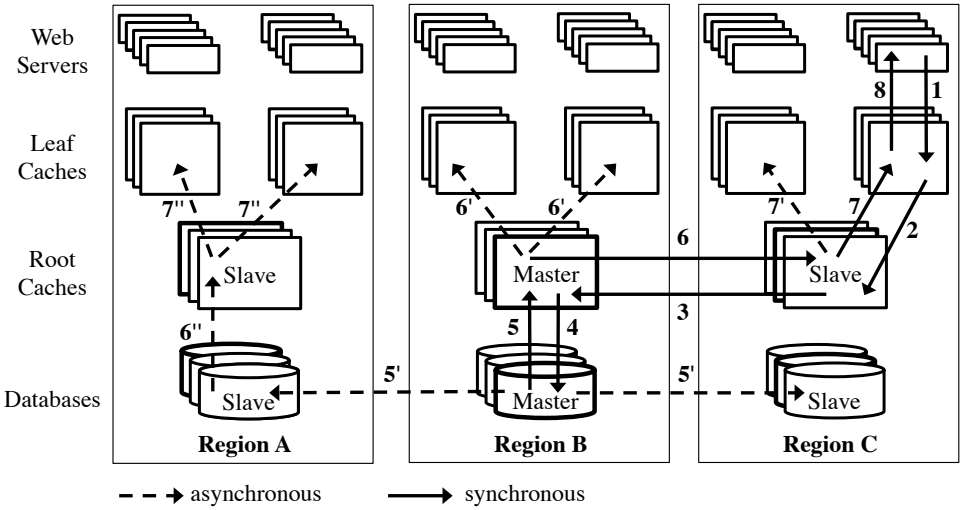


Figure 1: The write path for Facebook's replicated storage. On-path caches are synchronously updated; off-path caches are asynchronously invalidated. There is a single master region for each shard with the root-master cache and master database. Different shards may have masters in different regions.

and accessed by their tail (source). Some edge types are symmetric, i.e., an edge(x,y) is always accompanied by a symmetric edge(y,x). For instance, if user 450 follows user 520, this relationship is represented by a pair of symmetric edges: a following edge$(450,520)$ and a followed_by edge$(520,450)$. Symmetric edge types are identified in the schema. A creation/deletion operation on one edge in a pair implicitly creates/deletes the symmetric edge. For instance, creating the following edge$(450,520)$ would automatically create the followed_by edge$(520,450)$.

Database Facebook uses a horizontally (i.e., row) sharded, geo-replicated relational database management system to persistently store user data. There is a full copy of the entire database in each of the geo-distributed *regions*, which are co-located sets of datacenters. For each shard there is a single master region that asynchronously updates the other, slave regions. The master region is not necessarily the same for all shards and the master region for a shard can be changed.

Two-Level Cache A two-level cache fronts (i.e., it is a write-through cache) the full database replica in each region. There is a single logical *root* cache that sits in front of the database; there are multiple logical *leaf* caches that sit in front of the root cache.[2] Shards of root caches are distinguished by the role of the database shard they front: a *root-master* fronts the master for its shard of the database, a *root-slave* fronts a slave for its shard of the database. The front-end web servers that handle user requests are grouped into clusters and each cluster is exclusively associated with a single leaf cache. Figure 1 shows clusters of web servers, the top-level leaf caches, mid-level root cache, and bottom-level database in three regions.

[2] The root and leaf caches are also called *leader* and *follower* tiers [14].

Reads progress down the stack in their local region on cache misses from leaf cache to root cache, and then to local database. The cache-hit ratios are very high, so reads are typically served by the leaf caches.

Writes follow a more complex route as shown in Figure 1. They are synchronously routed through their leaf cache (1) to their local root cache (2) to the root-master cache (3), and to the master database shard (4) and back (5–8). Each of those caches applies the write when it forwards the database's acknowledgment back towards the client. The root caches in the master (6′) and originating regions (7′) both asynchronously invalidate the other leaf caches in their region. The database master asynchronously replicates the write to the slave regions (5′). When a slave database in a region that did not originate the write receives it, the database asynchronously invalidates its root cache (6″) that in turn asynchronously invalidates all its leaf caches (7″).

2.2 Consistency Models

This subsection reviews the definition of "local" for consistency models, the consistency models covered by our principled anomaly checkers, and the consistency model provided by Facebook's replicated storage.

Local Consistency Models A consistency model, *C*, is *local* if the system as a whole provides *C* whenever each individual object provides *C* [29]. We primarily study local consistency models in this paper because they can be checked and reasoned about using requests to a subset of objects. Analyzing non-local consistency models—e.g., strict serializability, sequential consistency, and causal consistency—require requests to all objects, and thus are not amenable to sampling-based study. While we focus on local consistency models, we do derive lower bounds, and also some upper bounds on the anomalies non-local consistency models would prevent in Section 3.4. All of the consistency models described in the remainder of this section are local.

Linearizability Linearizability [29] is the strongest consistency model for non-transactional systems. Intuitively, linearizability ensures that each operation appears to take effect instantaneously at some point between when the client invokes the operation and it receives the response. More formally, linearizability dictates that there exists a total order over all operations in the system, and that this order is consistent with the real-time order of operations. For instance, if operation *A* completes before operation *B* begins, then *A* will be ordered before *B*. Linearizability avoids anomalies by ensuring that writes take effect in some sequential order consistent with real time, and that reads always see the results of the most recently completed write. Linearizability also decreases programming complexity because it is easy to reason about.

Per-Object Sequential Consistency Per-object sequential consistency[3] [16, 34] requires that there exists a legal, total order over all requests to each object that is consistent with client's orders. Intuitively, there is one logical version of each object that progresses forward in time. Clients always see a newer version of an object as they interact with it. Different clients, however, may see different versions of the object, e.g., one client may be on version 100 of an object, while another client may see version 105.

Read-After-Write Consistency Within each cluster, Facebook's replicated storage is also designed to provide read-after-write consistency within each leaf cache, which means when a write request has committed, all following read requests to that cache always reflect this

[3] Also called per-record timeline consistency.

write or later writes. We also consider read-after-write consistency within a region and globally. Region read-after-write consistency applies the constraint for reads in the same region as a write. Global read-after-write consistency applies the constraint for all reads.

Eventual Consistency Eventual consistency requires that replicas "eventually" agree on a value of an object, i.e., when they all have received the same set of writes, they will have the same value. Eventual consistency allows replicas to answer reads immediately using their current version of the data, while writes are asynchronously propagated in the background. While writes are propagating between replicas, different replicas may return different results for reads. A useful way to think about this is that a write might not be seen by reads within time Δt after it committed, which is not allowed in linearizable systems. We later refer to this Δt as a *vulnerability window*. Multiple replicas may accept writes concurrently in an eventually consistent system, but we do not have that complication here, because Facebook's TAO system uses a single-master-per-shard model to order all writes to an object.

Facebook's Consistency Overall Facebook's design provides per-object sequential consistency and read-after-write consistency within a cache, and eventual consistency across caches. User sessions are typically handled exclusively by one leaf cache, and thus we expect most of them to receive per-object sequential and read-after-write consistency.

Sometimes user sessions are spread across multiple leaf caches. This happens when user sessions are load-balanced between web server clusters. It also happens when a machine within a leaf cache fails and the requests to it are sent to other leaf caches in the same region. In these cases we expect user sessions to receive eventual consistency [14].

3. Principled Consistency Analysis

This section makes progress towards quantifying the benefits of stronger consistency models by identifying how often the behavior they disallow occurs in Facebook's replicated storage. This section describes the trace collection, the consistency checkers, the results of those checkers, and the conclusions about the benefits of stronger consistency.

3.1 The Trace

We collect a trace that is useful for identifying the violations of consistency models. An ideal trace would contain all requests to all vertices (and their adjacent edges) in the system and would allow us to check all consistency models. Unfortunately, logging all requests to the replicated storage system is not feasible because it would create significant computational and network overhead. To avoid this, our trace instead contains all requests to a small subset of the vertices (and their adjacent edges) in the system. This trace allows us to check local consistency models, while keeping collection overhead low.

Trace Collection Vertices are identified by 64-bit unique ids. These ids are not necessarily evenly distributed so we hash them before applying our sampling logic. This ensures an even distribution among vertex types and a rate of logged requests that is similar to our sampling rate. We trace vertex requests if the hashed ID is in our sample. We trace edge requests if the hashed head or tail ID is in our sample. Tracing based on both directions of edge requests ensures we catch explicit requests that implicitly update the opposite direction for symmetric edge types.

For each request, we log the information necessary for running our checkers and debugging identified anomalies:

- ***object_id***: Vertex ID or head and tail IDs for edge requests.
- ***type***: The type of the vertex or edge requested.
- ***action***: The request type, e.g., vertex_create, edge_add.
- ***value***: The hashed value field of the request.
- ***invocation_time***: Time when the request was invoked.
- ***response_time***: Time when a response was received.
- ***user_id***: Hashed ID of the user that issued the request.
- ***cluster***: ID of the cluster that served the request.
- ***region***: ID of the region that served the request.
- ***endpoint***: ID of the service that issued the request.
- ***server***: ID of server that issued the request.

The object_id, type, and action fully define a request. The value allows us to match reads with the write(s) they observed. The invocation_time and response_time are used for defining the real-time order in linearizability and read-after-write consistency. Those times in combination with the user_id define the (process) ordering for per-object sequential consistency. The cluster is used to differentiate different varieties of anomalies by checking per-cluster, per-region, and global consistency. The endpoint and server are useful for debugging anomalies we identify.

Requests are logged from the web servers that issue the requests to the storage system. They are logged to a separate logging system that stores the requests in a data warehouse. At the end of each day we run the offline consistency checkers. Waiting until well after the requests have occurred to check them ensures we see all requests to an object. This eliminates any effects we would otherwise see if parts of the logging system straggle.

Clock Skew The web servers that log requests take the invocation timestamp before issuing each request, and the response timestamp after receiving the response. The time on the web servers is synchronized using NTP [40] and there is a small amount of *clock skew*, the amount of time a machine's local clock differs from the actual time. For the days of August 20-31 the 99.9^{th} percentile clock skew across all web servers was 35 ms.

We account for this clock skew by expanding the invocation time and response time, i.e., we subtract 35 ms from all invocation times and add 35 ms to all response times. This ensures all anomalies we identify are true anomalies, and thus we are certain that a system with a strong consistency model would return a different result. Another way to view this is that we are identifying a close lower bound on the true number of anomalies.

Coping with Imperfect Logging One tricky issue we dealt with in collecting and processing our trace was that our logging was not lossless. In particular, we log from web servers that prioritize user traffic over our logging and as a result sometimes cause it to timeout. This means the trace does not contain 100% of requests for our sample of objects. For reads, we do not believe this has much impact on our results because the missing reads should be evenly distributed between anomalous and regular reads.

For writes, however, missing them presents a problem that results in many false positive anomalies. Reads that reflect the results of a write not in the trace would be marked as anomalies. For instance, in:

```
vertex_write(450,"x:1",...)
vertex_write(450,"x:2",...) # missing
vertex_read(450, ...) = "x:2"
```

the read appears to be an anomaly in the trace because it reflects a state of the data store that never (appears to have) existed. We encountered many of these apparent anomalies,

including some cases where a value we never saw written was returned for hours. We investigated them by checking the state of the master database and confirmed that the reads were reflecting a write that was missing from our trace.

Our initial approach for dealing with these missing writes was to exclude reads that did not match with a write in our trace. This provided a better measurement, but we still saw some false positives. For instance, in:

```
vertex_write(450,"x:2",...)
# potentially many hours and operations
vertex_write(450,"x:1",...)
vertex_write(450,"x:2",...) # missing
vertex_read(450, ...) = "x:2"
```

the read still incorrectly appears to be an anomaly because it appears to be reflecting a too old state.

To eliminate these false positives from our trace we supplemented it with an additional trace of writes from the Wormhole system [48]. This secondary trace uses the same hashing and sampling logic to determine which writes to log. Its logging is also not lossless, so again we do not have 100% of writes to all objects in the sample. When we combine the writes from both traces, however, we have ~99.96% of all writes to the objects in our sample. The impact of the few remaining missing writes is negligible when we add our logic that identifies obviously missing writes to the combined trace.

3.2 Anomaly Checkers

We designed a set of algorithms to identify anomalies for three consistency models: linearizability, per-object sequential consistency, and read-after-write consistency. Consistency models provide guarantees by restricting the set of possible executions. The basic idea of these algorithms is to identify when a traced execution violates these restrictions, i.e., it is not possible in a system that provides the checked consistency guarantee. Each checker does this by maintaining a directed graph, whose vertices represent the state of an object, and whose edges represent the constraints on the ordering between them. We check for anomalies by checking that the state transition order observed by reads is consistent with these constraints.

Preprocessing We preprocess the trace to reduce its size for faster checking, to reduce the memory required by the checkers, and to simplify the implementations. Each consistency model that we check is a local consistency model, so we can check it by checking each object individually. Our first preprocessing step enables this by grouping the requests for each object together.

Our second preprocessing step reduces the size of the trace by eliminating objects that will not show anomalies. These eliminated objects include those that either have no writes, or have no reads. Objects with no writes have no restrictions on the set of allowed values. Objects with no reads have nothing to check against the set of allowed values.

Our final preprocessing step sorts the requests to each object by their invocation time.[4] This step simplifies the implementation of the checkers and allows them to make a single pass over the requests to each object.

[4] Sorting is primarily to merge traces from individual web servers. In addition, it also deals with occasionally delayed components in the logging infrastructure that can cause out of order logging.

```
# all requests to one object sorted by their invocation time
func linearizable_check(requests):
  for req in requests
    add_to_graph(req)
    if req is read
      # look ahead for concurrent writes
      next_req = req.next()
      while next_req.invocT < req.respT
        if next_req is write
          graph.add_vertex(next_req)
      match = find_matched_write(req)
      merge_read_into_write(req, match)
      if found_anomaly(graph)
        anomaly_reads.add(req)
    # graph only has writes and is acyclic
  print anomaly_reads

func add_to_graph(req):
  if req in graph
    # already in graph from lookahead
    return
  new_v = graph.add_vertex(req)
  # add edges from real-time ordering
  for v in graph.vertices()
    if v.resp_t < new_v.invoc_t
        graph.add_edge(v, new_v)
# matched write inherits edges read
func merge_read_into_write(read, match)
  for e in read.in_edges()
    if e.source != match
      graph.add_edge(e.source, match)
  # refine response time of merged vertex
  if req.resp_t < match.resp_t
    match.resp_t = req.resp_t
  graph.remove_vertex(read)

func find_matched_write(req)
  for v in graph.breadth_first_search(req)
    if v.hashValue matches req.hashValue
      return v

# cycles indicate no legal total order exists and this
#  read is an anomaly
func found_anomaly(graph)
  cycles = graph.check_for_cycle()
  if cycles is null
    return false
  # remove edges that produced cycles
  for c in cycles
    for e in c.edges()
      if e.source.invoc_t > e.dest.resp_t
        graph.remove_edge(e)
  return true
```

Linearizability Checker Figure 2 shows the pseudocode for a simplified version of our linearizability checker. The input to the checker is a list of all of the operations to one object sorted by invocation time. The output of the checker is all of the anomalous reads, i.e., reads that return results they would not in a linearizable system.

Intuitively, the checker maintains a graph whose vertices are operations, and edges are constraints. It checks for cycles as it adds operations to the graph. If the graph is acyclic, then there exists at least one total order over the operations with all constraints satisfied, i.e., there are no anomalies.[5] If there are cycles, then there are anomalies. After adding an operation to the graph we check for cycles; if there are cycles then the operation is flagged as an anomaly and the cycle broken. Breaking cycles maintains the invariant that the graph is acyclic before an operation is added, and thus allows us to check if the operation is an anomaly simply by checking for cycles after adding it.

Linearizability requires that there exists a total order that is legal, and agrees with the real-time ordering of operations. A total order is *legal* if a read is ordered after the write it observed with no other writes in between. Our checker enforces this constraint by merging a read vertex into the write vertex it observed (lines 29–36). This leaves us with only write vertices in the graph. We can then convert a total order of the write vertices into a total order of all operations by simply placing the reads immediately after the write they were merged into.

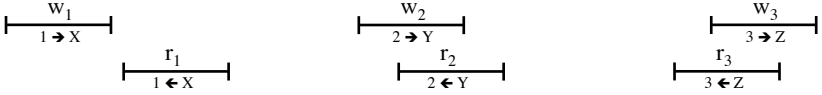
Merging read vertices into the write vertices they observe requires matching reads to writes (lines 38–41). Figure 3 shows the three possible cases: a read observes an earlier write (a), a read observes a concurrent write that began before it (b), and a read observes a concurrent write that began after it (c). Handling the third case when processing operations ordered by invocation time requires that we look ahead to find all the concurrent writes (lines 9–11, 20–22).

The real-time ordering constraint dictates that any operation that is invoked after another completes must be ordered after it. Our checker incorporates this constraint by adding edges to an operation from all those that precede it (lines 25–27). Figure 4 shows the execution for a *stale read anomaly* where the real-time ordering is violated when a read observes the value of an old write instead of a more recent one. The figure also shows the cycle this anomaly creates in the graph that our checker will catch (lines 14–15, 45–54).

The total order requirement of linearizability may still be violated even when reads do not observe stale values. This type of anomaly typically occurs when there are concurrent writes followed by reads that do not mutually agree on the execution order of the writes. We term this type of violation a *total order anomaly* and show an example in Figure 5. The figure also shows the cycle the anomaly creates in the graph that our checker will find and flag.

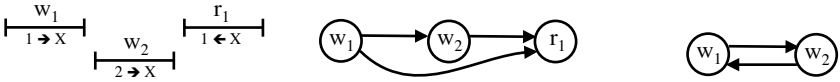
An important part of making our checker complete is refining the response time of a write vertex to be the minimum of its response time and the response times of all the reads that observe it (lines 34–35). This ensures we enforce transitive ordering constraints where a read must be after the write it observes. Operations that follow that read in real-time must be ordered after it, and thus operations that follow the read must also be after the write. For instance, our checker would refine the response time of w_3 in Figure 3 to be that of r_3 to ensure that if some other operation began after r_3's response time but before w_3's response time, we would still capture that constraint.

[5] A topological sort of the graph would give one such total order.



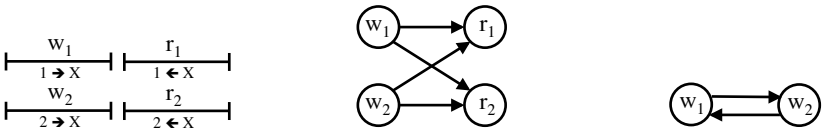
(a) Earlier write. **(b) Concurrent write, begins earlier.** **(c) Concurrent write, begins later.**

Figure 3: Reads observing writes.



(a) Execution. **(b) Graph before merge.** **(c) Graph after merge.**

Figure 4: Stale read anomaly. Bars indicate invocation and response times. Edges indicate a real-time ordering constraint. r_1 is merged into w_1 from (b) to (c).



(a) Execution **(b) Graph before merge.** **(c) Graph after merge.**

Figure 5: Total order anomaly. r_1 and r_2 are merged into w_1 and w_2 respectively.

For clarity we omit pseudo-code for a number of corner cases and optimizations. The corner cases include leading reads, missing writes, and picking between multiple sets of reads for concurrent writes. Leading reads are the reads for an object in the trace that occur before we see any writes. We handle these reads by inserting special ghost writes into the beginning of the trace for the writes that we can assume happened, but were before our trace began. Missing writes are the writes that are still missing from the merged trace. We handle these as we did before we collected a second trace. When there are total order anomalies there can be multiple sets of competing reads. Only one set can be marked as not anomalous. Instead of picking the set of the first write to be seen, like the pseudo-code, we pick the largest set, i.e., we assume the majority of the reads correctly reflect the final state of those concurrent writes.

The optimizations we omit include reducing the number of edges and how we check for cycles. Our implementation only adds edges that express new constraints, e.g., in Figure 4(b) we would not add the edge (w_1, r_1) because that constraint is already expressed by the edges (w_1,w_2) and (w_2,r_1). We check for cycles by checking if a read is being merged into a write that is its ancestor (stale read anomalies) and checking each set of concurrent writes to ensure no more than one is observed by reads invoked after the set returns (total order anomalies).

Per-Object Sequential and Read-After-Write Checkers Linearizability is strictly stronger than per-object sequential and read-after-write consistency, meaning the anomalies in linearizability are supersets of those in the weaker models. We exploit this property by building the weaker model checkers as add-on components to the linearizability checker. Each operates only on the linearizable anomalies.

Per-object sequential consistency has two constraints on the requests to each object: the order of requests is consistent with the order that the users issue their requests and there

	Requests			Objects
	Total	Reads	Writes	
Vertices	939	937	2.1	3.4
Edges	1,828	1,818	9.7	13.4

Table 1: High-level trace statistics in millions. Objects indicate the number of unique groups of requests that our checkers operate over. For vertices it is the count of unique vertices in the trace. For edges it is the count of unique source and edge type combinations.

exists a total order. We check these two constraints against each linearizable anomaly by checking the following conditions that reflect those two constraints respectively. Case 1: the linearizable anomaly is a stale read anomaly, and there exists a write more recent than the matched write that is from the same user who issued the read. Case 2: the linearizable anomaly is a total order anomaly. The former case shows a client does not observe her most recent write, and hence is inconsistent with client's order. The latter case is precisely a total order violation. If either of these two cases matches, then the linearizable anomaly is also flagged as a per-object sequential anomaly.

Read-after-write consistency requires that all reads after a committed write always reflect this write or later writes. This constraint is equivalent to the real-time ordering constraint. Hence, our checker simply flags all stale read anomalies found by linearizable checker as read-after-write anomalies. We check read-after-write violations in three levels: cluster, region, and global. We check at the cluster/region level by looking for a write in the cycle that is more recent than the matched write, and has the same cluster/region as the read. Global read-after-write anomalies are the same as the stale read anomalies under linearizability.

3.3 Analysis

We ran our consistency checkers on a 12-day-long trace of Facebook's replicated storage. Each day is checked individually, and then the results are combined. The trace was collected from August 20, 2015 to August 31, 2015 and included all requests to 1 out of every million objects. This sampling rate is low enough to avoid adversely impacting the replicated storage system while being high enough to provide interesting results. The trace contains over 2.7 billion requests. Table 1 shows more high-level statistics.

	No Writes	No Reads	Both
Vertices			
Objects	75.8%	13.5%	10.7%
Requests	74.2%	0.1%	25.7%
Edges			
Objects	74.3%	21.6%	4.1%
Requests	76.2%	1.0%	22.8%

Table 2: Percentage of objects with no writes, no reads, and with both reads and writes. The percentage of overall requests to objects of each category is also shown. Only requests to objects that contain both reads and writes can exhibit anomalies.

	Anomalous Reads	Percentage Of Filtered (241M)	Overall (937M)
Linearizable	3,628	0.00151%	0.00039%
Stale Read	3,399	0.00141%	0.00036%
Total Order	229	0.00010%	0.00002%
Per-object Seq	607	0.00025%	0.00006%
Per-User	378	0.00016%	0.00004%
Read-after-Write			
Global	3,399	0.00141%	0.00036%
Per-Region	1,558	0.00065%	0.00017%
Per-Cluster	519	0.00022%	0.00006%

Table 3: Anomalies for vertices. Filtered reads are those remaining after preprocessing.

Preprocessing Results To reduce the computational overhead as well as to better understand our workload, we pre-process the trace to filter out objects whose requests follow patterns that would never show anomalies. The two patterns of requests we exclude are all reads and all writes. If an object has only reads, then there are no updates for the storage system to propagate around, and thus no possible anomalies. If an object has only writes, then there are no reads that could be marked as anomalies. Table 2 shows the preprocessing results. This preprocessing quickly yields upper bounds of 25.7% of requests to vertices and 22.8% of requests to edges that can exhibit anomalies.

Anomalies in Vertices Table 3 shows the results of our checkers for requests to vertices. We see a very low percentage of requests to vertices, around 0.00039%, violate linearizability. Most of these anomalies are stale read anomalies, i.e., the read did not see the most recent update. A smaller number of them are total order anomalies.

The stale read anomalies are identified in the stale read row, the per-user row under per-object sequential, and all of the read-after-write rows. The source of stale reads is typically replication lag that includes master to slave wide-area replication and asynchronous cache invalidations up the cache hierarchy. While this replication is still ongoing a read may return an older, i.e., stale, version that it would not in a system with stronger consistency. The replication lag can thus be considered the *vulnerability period* during which anomalies may occur. The effect of increasing replication lag, and thus an increasing vulnerability period is shown in the read-after-write results. Replication lag increases from the cluster to the region and to then global level and we see an increase from 519 to 1,558, and to 3,399 anomalies.

Total order anomalies are identified in the total order row. They contribute to the overall linearizability and per-object sequential counts. The source of these anomalies is also typically replication lag. In this case, however, multiple users are reading different versions of an object, i.e., one user in the same cluster as a recent write is reading the new version and one user in a different cluster is reading an old version.

In general, the low percentage of anomalies is primarily due to the low frequency of writes, and the locality of requests to an object. Both of these factors decrease the likelihood of having a read occur during the vulnerability window after a write. The low frequency of writes—i.e., only 1 in 450 operations was a write—directly decreases the likelihood

	Anomalous Reads	Percentage Of Filtered (417M)	Overall (1,818M)
Linearizable	12,731	0.00305%	0.00070%
Stale Read	9,831	0.00236%	0.00054%
Total Order	2,900	0.00070%	0.00016%
Per-object Seq	2,900	0.00070%	0.00016%
Per-User	0	0%	0%
Read-after-Write			
Global	9,831	0.00236%	0.00054%
Per-Region	5,312	0.00127%	0.00029%
Per-Cluster	3,070	0.00074%	0.00017%

Table 4: Anomalies for edges. Filtered reads are those remaining after preprocessing.

a given read will be during a vulnerability period. The locality of requests to an object also decreases the likelihood of anomalous behavior because Facebook's clusters provide read-after-write consistency, i.e., there is no vulnerability window within a cluster. Our results mostly validate this claim as we see approximately 1 in 1.8 million requests not receive per-cluster read-after-write consistency. The few exceptions are likely due to cache machine failures.

Anomalies in Edges Table 4 shows the results of our checkers for requests to edges. The rate of anomalies is doubled compared to the anomaly rate of vertices, yet it is still a very small number—we only observe 1 linearizability anomaly out of 150,000 requests. The higher frequency of edge anomalies is correlated to the higher frequency of write operations on edges. As shown in Table 1, there is 1 write in every ~188 operations on edges, while the write fraction of vertex operations is about 1 out of 450. Intuitively, more updates mean more frequent changes in system states. Each write introduces a vulnerability window that allows anomalous reads to happen.

The rate of the different types of anomalies for edges is double what we observed for vertices, with the notable exception of total order violations, and per-user session violations. We see a rate of total order violations that is more than ten times the rate we see with vertices and we see no per-user session violations. We are still investigating the root cause of both of these differences.

Figure 6 further breaks down anomalies by the type of the edge. It shows a CDF of anomalies for types in rank order, i.e., the type with the most anomalies is at position 1. The rank 1 edge type contributes ~60% of all observed anomalies. This is the "like" edge type, which is frequently updated and requested. The high update and request rate of "likes" explains their high contribution to the number of overall anomalies. The high update rate induces many vulnerability windows during which anomalies could occur and the high request rate increases the likelihood a read will happen during that window. The top 10 types together account for ~95% of the anomalies. These most-anomalous edge types have implications for the design of systems with strong consistency, which we discuss below, and for finding programmer errors, which we discussion in Section 5.

Upper Bound Our analysis is primarily concerned with identifying reads that would definitely return different results in systems with stronger consistency. As a result, when there is uncertainty due to clock skew we err on the side of reporting fewer anomalies

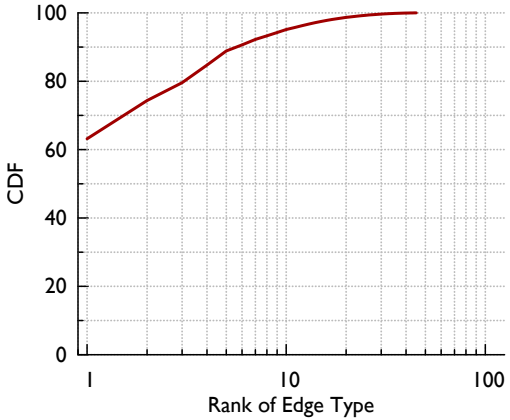


Figure 6: This figure shows the CDF of anomalies by edge type in ranked order. For instance, the rank 1 edge type has the highest number of anomalies.

and so our results are a lower bound on the effect of stronger consistency models. In our main results we have expanded the invocation and response times of each request by 35ms (99.9^{th} percentile clock skew). Figure 7 shows the effect of different choices for expanding the invocation and response times for vertices.

An expansion of -35ms gives an upper bound on the effect of the stronger consistency models. Here 35ms is added to the invocation time, 35ms is subtracted from the response time, and we limit the response time to be no earlier than the modified invocation time. We see anomaly rates that are much higher for both -35ms and -17.5ms. This is because in both cases an artificial and incorrect ordering of requests is being enforced. The artificial ordering comes from the invocation times of the many requests that completed in less than 70ms/35ms and thus had their invocation to response time windows shrunk to 0. In reality these requests were concurrent and could have been ordered either way, not only in their invocation time order.

The results for 0ms give a more accurate view of the true rate of anomalies because they mostly avoid artificially ordering concurrent requests. With no time expansion we see 0.00066% linearizable anomalies, 0.00008% per-object sequential anomalies, 0.00065% global read-after-write anomalies, 0.00029% region read-after-write anomalies, and 0.00010% cluster read-after-write anomalies.

3.4 Discussion

Our principled analysis gives us insight into the effect of deploying replicated storage with stronger consistency at Facebook. The primary benefits of stronger consistency are the elimination of anomalous behavior that is confusing to users and a simplified programming model. Our results here indicate how often anomalies occur, i.e., how often a stronger consistency model could help.

Quantifying the Benefit of Stronger Consistency Figure 8 shows a spectrum of consistency models and the rate of anomalies for requests in them. Our observed values for linearizability, per-object sequential consistency, and read-after-write consistency are shown in the spectrum. In addition, the relationship between different consistency models is shown

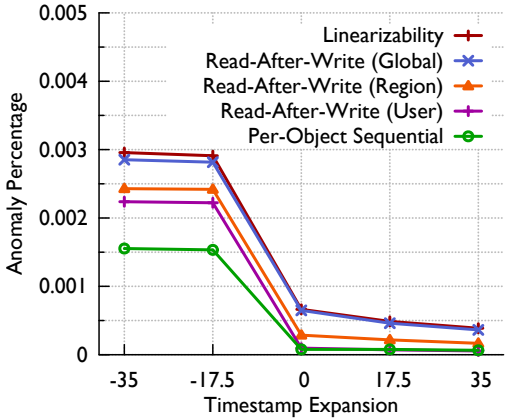


Figure 7: The percentage of reads flagged as anomalies by our checkers for different timestamp expansions for vertices. Expanding timestamps Xms subtracts X from the invocation time and adds X to response time.

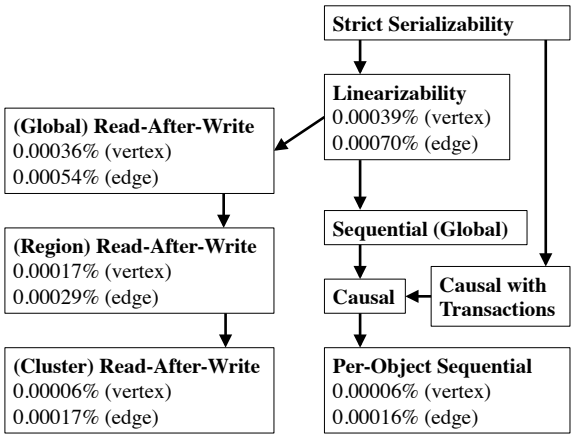


Figure 8: Spectrum of consistency models with arrows from stronger to weaker models. Measured anomaly rates are shown. Bounds on the anomaly rate in unmeasureable non-local consistency models can be inferred.

by arrows pointing from a model A to another model B when A is is strictly stronger than B.[6]

These relationships allow us to bound the effect of non-local consistency models that we cannot write checkers for, namely (global) sequential consistency [34], and causal consistency [1, 33]. For instance, because causal is stronger than per-object sequential it would eliminate anomalies from at least 0.00006% of vertex requests and because it is weaker than linearizability it would eliminate anomalies from at most 0.00039% of vertex requests.

[6] See Section 4.5 for the definition of strictly stronger.

We also show strict serializability and causal consistency with transactions in the spectrum. Our results give lower bounds for the anomalies each of these models would prevent. We cannot provide upper bounds for them, which we discuss further as a limitation later in this section.

An Interesting Direction Another result of the analysis was identifying that a small number of edge types account for the vast majority of anomalies. This points to two interesting directions for future research into systems with stronger consistency: (1) build system were non-anomalous types have negligible overhead or (2) provide stronger consistency for a small subset of a larger system. While the latter would not prevent all anomalies, it would allow incremental deployment of these systems and significantly reduce the rate of anomalies. Interestingly, such a subsystem that does provide linearizability is used within Facebook for a small set of object types, e.g., passwords.

Limitations Our analysis has a few limitations to what it can provide in terms of quantifying the benefits of stronger consistency. One limitation is that our analysis is limited to the replicated storage at Facebook. Results for other eventually consistent systems could and would be different, but Facebook is a large, important data point in this space.

Another limitation is that our measurements only report anomalies we observed, not anomalies that could happen but whose triggering conditions have not yet occurred. An architectural change could trigger these anomalies, but our principled anomaly checkers would alert us to the effects of this change. In addition, it is likely that our practical consistency checker would catch this effect in real-time. Our sampled view of objects is also a limitation. There might be rare objects with higher rates of inconsistencies that our sampling misses that could shift the overall anomaly rates.

The biggest limitation to our analysis is that it cannot give insights into the benefits of transactional isolation. Transactional isolation is inherently a non-local property [29] and so we cannot measure it accurately using only a sample of the full graph. This unfortunately means we cannot quantify the benefits of consistency models that include transactions, e.g., serializability [44] and snapshot isolation [11], or the benefit of even read-only transactions on other consistency models. For instance, while our results for causal consistency bound the benefit of the COPS system [38], they do not bound the benefit of the COPS-GT [38] system that also includes read-only transactions.

4. Practical Consistency Analysis

Our practical consistency analysis has been used since 2012 at Facebook as a real-time cache-health monitoring system. This section justifies why we need practical analysis, defines ϕ-consistency, describes typical measurements of it, discusses their implications, and finally describes how we use ϕ-consistency to debug production issues.

4.1 Why We Need Practical Analysis

Our principled analysis is helpful for understanding the consistency guarantees the current systems provide, identifying consistency issues caused by weak consistency models, and quantifying the benefits of a system with stronger consistency. However, the principled analysis is neither designed for real-time monitoring nor is a good fit. The principled analysis requires access to all timestamped requests to each sampled object. Retrieving and analyzing these requests in real-time would be akin to implementing a replicated storage system with strong consistency. Our principled analysis avoid this overhead by only processing requests well after they have occurred, typically once per day. This allows

the storage for the principled analysis trace to be eventually consistent, and provides plenty of time for log entries to arrive.

In contrast, our practical consistency analysis is designed to operate in real-time and to be lightweight. As a consequence it does not trace all operations on a given object, and thus does not give insights into how often principled consistency models are violated. Instead, it uses injected reads to track metrics that are designed to mirror the health of different parts of the replicated storage.

4.2 $\phi(P)$-consistency

The operational consistency metric we use is $\phi(P)$-consistency. The $\phi(P)$-*consistency* of a set of replicas P is the frequency that injected reads for the same data to all $p \in P$ receive the same response from each p.

At Facebook our replicas can be a leaf cache, root cache, or database. When performing a read for $\phi(P)$-consistency, we care about the data at the individual $p \in P$. Reads are made with a flag to look only in cache for a cache layer, avoiding any read-through nature of the system. A read-miss indicates there is no cached data at p, so it is not considered in the metric.

The injected reads are all issued from one server and the responses are compared once all have returned. This does not require clock synchronization, versioning, or logging on either the issuing client or the replicated storage, and is very lightweight. As a result, however, it is affected by network and other delays. For instance, even if two replicas of an object have the same old version of it when the reads are issued, and change to the same new version at the same instant in time, if one of the injected reads arrives before this instant and the other afterwards, they will be $\phi(P)$-inconsistent. Despite this and its theoretical incomparability to principled consistency models that we will show in Section 4.5, $\phi(P)$-consistency is useful in practice.

$\phi(P)$-consistency's usefulness derives from how it quickly approximates how convergent/divergent different parts of the system are. Increases in network delay, replication delay, misconfiguration, or failures all cause a drop in $\phi(P)$-consistency. An increase in the write rate of the system will also cause a drop in $\phi(P)$-consistency rate, because there will be more writes in flight at any given time. These types of changes can be detected within minutes by our ϕ-consistency metrics.

We track two types of $\phi(P)$-consistency regularly: $\phi(G)$-consistency and $\phi(R_i)$-consistency. $\phi(G)$-*consistency* is for the global set G of all replicas, i.e., all leaf and root cache clusters at Facebook. As a global system measure it is useful for tracking the health of the overall system. $\phi(R_i)$-*consistency* is for the set of all cache clusters in region R_i. We track $\phi(R_i)$-consistency for all regions. It is useful for tracking the health of each region and cluster individually.

4.3 Analysis

The practical analysis system that measures $\phi(P)$-consistency has been responsible for monitoring consistency issues since it was deployed in 2012. This subsection describes some results of this practical analysis.

Since ϕ-consistency rates are affected by the rate of updates to data, we track ϕ-consistency rates for several types of data with different access patterns. We expect $\phi(G)$-consistency to be lower than the $\phi(R_i)$-consistency because all results that count against any $\phi(R_i)$ also count against $\phi(G)$, and because $\phi(G)$ measures across geographically distant regions. Both rates exhibit diurnal and weekly patterns that correspond to the level of user

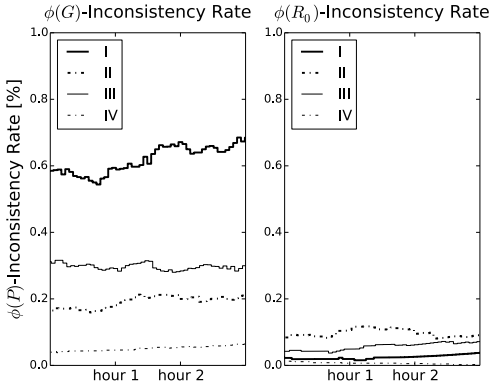


Figure 9

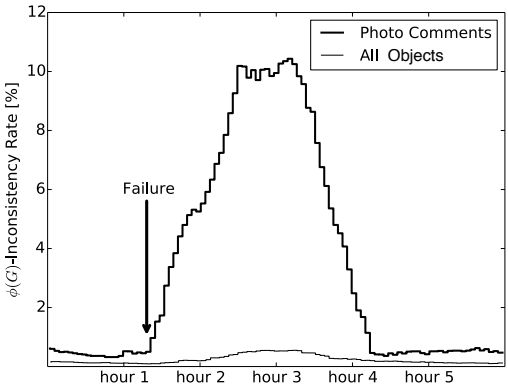


Figure 10

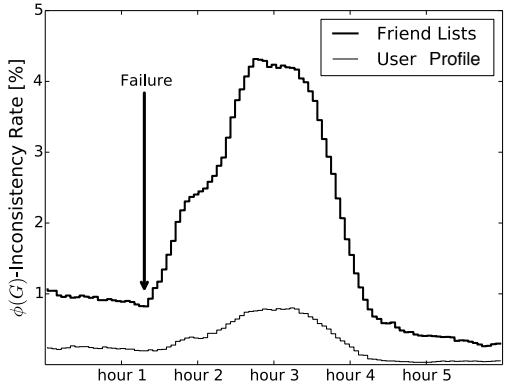


Figure 11

Figure 9 show $\phi(G)$-inconsistency, on the left, compared to $\phi(R_0)$-inconsistency (for the region R_0), on the right. Four types of data with varying sensitivity to inconsistency are shown. Figure 10 shows photo comment objects are sensitive to delays in the invalidation delivery pipeline. Figure 11 shows friend list requests are also sensitive to delays.

activity that occurs at Facebook. However, the difference between peak and trough is not significant.

Figure 9 shows $\phi(G)$-inconsistency and $\phi(R_0)$-inconsistency rates for four different types of data. The $\phi(G)$-inconsistency rate for Type I is significantly higher than the other three types. This is because Type I happens to be a type of data that changes frequently due to user activity. Fortunately, the $\phi(R_0)$-inconsistency of Type I data is much lower than its $\phi(G)$-inconsistency. Because users tend to consistently be served out of the same region, any inconsistencies tend to be undetectable by users.

One of the main goals in operating a system is to be able to quickly identify and respond to operational problems. Issues such as misconfiguration, network failures, and machine failures are all reasons that can cause performance to suffer. In practice, a single global measure of $\phi(P)$-consistency is insufficiently sensitive. Instead, we use $\phi(R_i)$-consistency rates to give us insight into problems that occur within individual regions. $\phi(R_i)$-consistency rates can increase for a single region indicating that there is an issue isolated to a specific region, this is shown in Figure 13 in Section 5 where we discuss an example issue. In addition, from experience, we have found that certain types of data are more sensitive to operational issues than others. We use their $\phi(G)$-consistency levels as early warning systems.

One of the sensitive families of data we have found are the objects that store photo comments. Intuitively, this makes sense because of user behavior. When a user uploads a photo, their friends may comment on the photo in interactive conversations. These conversations trigger a series of changes on the same object, which will need to be invalidated multiple times over a relatively short time period. The same object is read by all users who want to view the photo. This effect is exaggerated by users who have a large social graph, and those users' photos, in particular, produce highly consistency-sensitive objects.

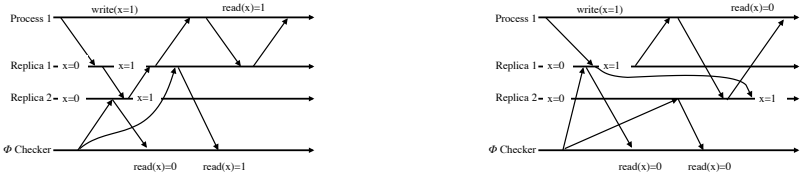
Figure 10 displays the spike in $\phi(G)$-inconsistency rate for photo comment keys after a site event that led to a substantial delay in site-wide invalidations. The rate eventually stabilized at 10%, meaning that about 10% of all photo-comment requests had a chance of returning different values depending on which caching tier the user was assigned to. The plot also shows the $\phi(G)$-inconsistency rates for an aggregation of all objects("All Objects") is far less sensitive than photo comment keys.

The invalidation slowdown depicted in Figure 10 also created problems for another type of object. Figure 11 shows the $\phi(G)$-inconsistency rate for two different types of requests: user profiles and friend lists. The user profile request reads a user's name and birthday. The more sensitive friend list request pulls out a user's friend list. Not surprisingly, the friend list request is more sensitive to invalidation delays because friend lists are modified far more often than a user's basic information.

4.4 $\phi(S:P)$-consistency

$\phi(P)$-consistency is inherently an aggregate concept, in that it measures consistency over the collection of $p \in P$. When there are inconsistencies, $\phi(P)$-consistency does not identify where the problem is. For example, if a single cache tier c always returns erroneous values, both $\phi(G)$-consistency and $\phi(R_i)$-consistency where $c \in R_i$ will be 0%.

To give finer granularity to our measurements we also use $\phi(S:P)$-consistency. $\phi(S:P)$-*consistency* is defined over two sets of replicas, S and P, and is the frequency with which requests to $s \in S$ return a value that is identical to the most common value returned by requesting all $p \in P$. This is a generalization of $\phi(P)$-consistency as that is equivalent to



(a) Linearizable but not ϕ-consistent execution. (b) ϕ-consistent but not linearizable execution.

Figure 12: Executions that demonstrate that ϕ-consistency and principled consistency models are incomparable.

$\phi(P:P)$-consistency. We often monitor $\phi(c_i:G)$-*consistency* for all cache tiers c_i. $\phi(c_i:G)$-consistency is not affected by errors from cache tiers $t \in G$ such that $t \neq c_i$. This makes it straight-forward to identify problematic cache tiers. In practice, $\phi(c_i:G)$ is especially useful for helping debug consistency issues.

The $\phi(P)$-consistency checker monitors inconsistency rate in real time and alarms when there is a spike in inconsistency. These alarms are often the first indication of a problem. Engineers use the $\phi(P)$-consistency checker together with other consistency monitoring systems to further investigate root causes. We consider a study of these systems, and how they work interactively to be interesting future work.

4.5 Where Principles Meet Practice

Consistency models can be compared theoretically by examining executions that are legal in one consistency model but illegal in the other. For two consistency models, A and B, if there is an execution that is acceptable in A but not B we say A is *weaker* than B. Intuitively, A is weaker because it allows behavior that B prohibits. If A is weaker than B, but B is not weaker than A then we say B is *strictly stronger* than A. For instance, linearizability is strictly stronger than per-object sequential consistency.[7] If A is weaker than B and B is weaker than A, then A and B are *incomparable*. Intuitively, this is because each permits behavior the other prohibits.

Linearizability and ϕ-consistency are incomparable because there are executions that are linearizable but not ϕ consistent and vice-versa. We show this through example executions in both directions. Figure 12a shows an execution that is linearizable but not ϕ-consistent and Figure 12b shows the reverse direction.

ϕ-consistency is also incomparable with per-object sequential and read-after-write consistency. Because linearizability is strictly stronger than per-object sequential and read-after-write consistency, all linearizable execution also satisfy the other models. Thus Figure 12a suffices to show they are weaker than ϕ-consistency. The careful reader will note that Figure 12b also shows the reverse direction of this relationship as the ϕ-consistent execution is neither per-object sequential nor read-after-write consistent.

5. Experience and Errors

This section qualitatively explores the impact of the increased programming complexity of eventual consistency. It describes why we saw few anomalies, common programmer errors,

[7] There are executions that are per-object sequential but not linearizable [29]. And there are no execution that are linearizable but not per-object sequential by definition because the real-time order requirement in linearizability implies the process order requirement in per-object sequential.

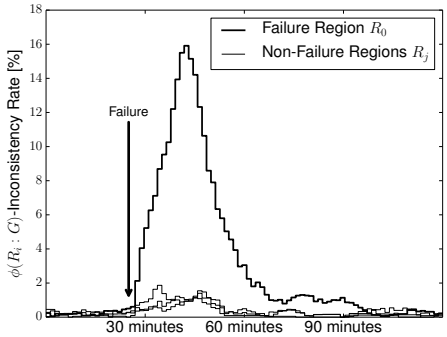


Figure 13: $\phi(R_i : G)$-inconsistency for the photo comment type after a misconfiguration. This metric enables us identify the geographic region to which the incorrect configuration was pushed.

and anti-patterns we teach developers to avoid that are common mistakes for uneducated programmers.

5.1 Few Anomalies

Our principled consistency analysis found fewer anomalies than we initially expected. Upon reflection, we surmise that it is due primarily to sensitive applications avoiding eventually consistent TAO. Sometimes sensitive applications—e.g., the login framework—mark reads as "critical" in TAO, which then provides linearizability by forcing requests to go through the master-root cache [14]. Other times, sensitive applications will build their own infrastructure to exactly match their consistency needs.

5.2 Human Errors

The most common root cause of problems we observe is human error, usually in the form of a misconfiguration. We alleviate impact of such issues by (1) limiting the damage that can be caused by a single misbehaving system and (2) quickly detecting and remediating issues when they occur.

Our web service deploys a large cache. In mid-2013, a bad configuration in the system that determines which cache machine to send a request to was deployed. This misconfiguration implied that cache invalidations generated for cached data were delivered to the wrong machine. Figure 13 shows the $\phi(R_i : G)$-inconsistency rates for photo comment keys for each region during this event. Note that the absence of invalidations only affected the 15% of users served out of the problem region. The minor spike in $\phi(R_j : G)$-inconsistency rate is a result of the fact that the majority value may sometimes be the stale value that is cached in the failure region, if the data item is not commonly cached in other regions. Within the problematic region, users are protected by local deletes which were still fully functional. In other words, the only way a user would notice a problem is by recognizing that the content originating from other regions are not reflected in their requests.

5.3 Developer Error

Our focus throughout the paper has been on our write-through cache that writes updated data into the database and that ensures the cache reflects the latest updates. We also have a look-aside cache for use cases where the write-through cache is not a good fit. The look-aside cache requires developers to manually write data to the database and

issue invalidations to the cache. In our experience, the look-aside cache is far harder for programmers to reason about. We next describe several common errors we observe in programs that use a look-aside cache. We use monitoring and developer education to identify and fix these issues.

- **Caching failures.** The data fetching logic might return an error code or failure of some type if a network disconnection or database request timeout occurs. If a program does not validate the data it receives, an error code or empty result will be stored in the cache. This pollutes the cache, and causes subsequent queries that could have succeeded to also return failure.
- **Negative caching.** Another common error is what we term *negative caching* where a program caches the lack of existence of data to save itself an expensive synchronous request. However, given the look-aside nature of some caches, when the underlying data is modified, this negative cache value has to be explicitly deleted, so a future read can demand fill the updated value. It is a common error to forget to add invalidations for the negatively cached data.
- **Caching time-dependent results.** Some queries to the database may give different results depending on the time the request was issued. One such request, for instance, would be "what site-wide announcements have not yet expired?" A common error is to cache a time-dependent result, and then reuse that result at a later time.
- **Caching cached results.** Another common error is for program code to cache data that is derived from another cached object–since the invalidation system is only aware of the first level of cached objects, the program code will often read stale data. This also decreases the available storage space in cache due to duplicate data.

6. Related Work

We review related work that measures the consistency of replicated storage, and work that specifies the consistency model of production systems.

Measuring Consistency Many benchmarking tools now include components that try to measure some form of the consistency provided by the data stores they are benchmarking. These systems include YCSB++ [45], BG [10], Wada et al. [53], Anderson et al. [4], Rahman et al. [47], Zellag et al. [55], Goleb et al. [28], and YCSB+T [20]. The most closely related work to ours is from Anderson et al. [4], which also takes traces of operations and runs offline checkers against them. Their checkers check safety, regularity, and atomicity violations [35]. Atomicity is equivalent to linearizability, regularity is equivalent to read-after-write consistency, and safety is an unusual relaxation of read-after-write consistency that permits an arbitrary value to be returned when there are concurrent writes. Our work builds on the checkers and methodology of these benchmarking tools.

All of these benchmarking tools generate a synthetic workload, collect a global trace, and then measure inconsistencies in that global, synthetic trace. In contrast, our work examines sampled, real traces. Using sampled traces allows us to analyze a production system running at large scale. Using real traces gives us insights into what anomalies occur in the current eventually consistent system, and what the benefits of stronger consistency would be.

Amazon measured how often their eventually consistent Dynamo [19] system returned multiple (concurrently written) versions of a shopping cart within a 24 hour period and saw that 99.94% of requests saw one version. Multiple versions are possible because Dynamo

uses sloppy quorums for writes that may not always intersect. This type of divergence is avoided by design at Facebook where there is a single master per shard that serializes all updates. Our work measures a different aspect of eventual consistency by looking at violations of many consistency models instead of divergence.

Probabilistically Bounded Staleness (PBS) [8] provides expected bounds on staleness for replicated storage that uses Dynamo-style sloppy quorums. It parameterizes a model based on replication delays and uses that to predict how often reads will return stale values and how stale those values will be. Our work was partially inspired by PBS, which is limited to sloppy quorums, is based on synthetic models, and only considers PBS k-staleness and PBS monotonic reads. In contrast, our work looks at a production single-master-per-shared system, is based on real measurements, and considers many principled and practical consistency models.

Production System Consistency Models Several publications from industry have explained the consistency model their systems provide. We were informed by them and measure anomalies under most of these models. Amazon's Dynamo [19] provides eventual consistency, this is the model Facebook's system provides globally. Google's Spanner [17] provides strict serializability. We measure anomalies under linearizability, which is a special case of strict serializability without transactions that provides a lower bound. Facebook's Tao system [14, 30] provides read-after-write consistency and Yahoo's PNUTS system [16] provide per-object sequential consistency (also called per-record timeline consistency). We measure anomalies under both models.

7. Conclusion

This paper studied the existence of consistency in the results from Facebook's TAO system and took a first step towards quantifying the benefits of stronger consistency in a large, real-world, replicated storage system. Our principled analysis used a trace of all requests to a sample of objects to study local consistency models. It identified when reads in TAO return results they would not in systems with read-after-write consistency, per-object sequential consistency, and linearizability. One key finding was that TAO is highly consistent, i.e., 99.99% of reads to vertices returned results allowed under all the consistency models we studied.

Another key finding was that there were anomalies under all of the consistency models we studied. This demonstrates that deploying them would have some benefit. Yet, we also found that these anomalies are rare. This suggests the overhead of providing the stronger models should be low for the trade-off to be worthwhile [2].

Our principled analysis used the relationship between consistency models to infer bounds on the benefits of the non-local consistency models that we could not directly study, e.g., sequential consistency. A key takeaway from this reasoning was that we could determine lower bounds, but not upper bounds, on the benefit of consistency models that include transactions, e.g., strict serializability or causal consistency with transactions. This suggests work on providing stronger consistency should include transactions to maximize its potential benefit.

Our practical consistency monitoring system tracks ϕ-consistency, a new consistency metric that is ideally suited for health monitoring. We showed that ϕ-consistency is theoretically incomparable to traditional consistency models, and that it is useful for an early warning and monitoring system. In addition, we gave insight into the effects of increased programming complexity caused by weaker consistency by discussing bugs our monitoring system has uncovered, and anti-patterns we teach developers to avoid.

Acknowledgments We are grateful to the anonymous SOSP reviewers for their extensive comments that substantially improved this work.

This work was supported by funding from the National Science Foundation Award CSR-1464438 (CRII).

References

[1] AHAMAD, M., NEIGER, G., KOHLI, P., BURNS, J., AND HUTTO, P. Causal memory: Definitions, implementation, and programming. *Distributed Computing 9*, 1 (1995).

[2] AJOUX, P., BRONSON, N., KUMAR, S., LLOYD, W., AND VEERARAGHAVAN, K. Challenges to adopting stronger consistency at scale. In *HotOS* (2015).

[3] ALMEIDA, S., LEITAO, J., AND RODRIGUES, L. Chainreaction: a causal+ consistent datastore based on chain replication. In *EuroSys* (2013).

[4] ANDERSON, E., LI, X., SHAH, M. A., TUCEK, J., AND WYLIE, J. J. What consistency does your key-value store actually provide? In *HotDep* (2010).

[5] ATTIYA, H., AND WELCH, J. L. Sequential consistency versus linearizability. *ACM TOCS 12*, 2 (1994).

[6] BAILIS, P., FEKETE, A., GHODSI, A., HELLERSTEIN, J. M., AND STOICA, I. Scalable atomic visibility with RAMP transactions. In *SIGMOD* (2014).

[7] BAILIS, P., GHODSI, A., HELLERSTEIN, J. M., AND STOICA, I. Bolt-on causal consistency. In *SIGMOD* (2013).

[8] BAILIS, P., VENKATARAMAN, S., FRANKLIN, M. J., HELLERSTEIN, J. M., AND STOICA, I. Probabilistically bounded staleness for practical partial quorums. *VLDB* (2012).

[9] BAKER, J., BOND, C., CORBETT, J. C., FURMAN, J., KHORLIN, A., LARSON, J., LEON, J.-M., LI, Y., LLOYD, A., AND YUSHPRAKH, V. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR* (2011).

[10] BARAHMAND, S., AND GHANDEHARIZADEH, S. BG: A benchmark to evaluate interactive social networking actions. In *CIDR* (2013).

[11] BERENSON, H., BERNSTEIN, P., GRAY, J., MELTON, J., O'NEIL, E., AND O'NEIL, P. A critique of ANSI SQL isolation levels. In *ACM SIGMOD Record* (1995).

[12] BERNSTEIN, P. A., AND GOODMAN, N. Concurrency control in distributed database systems. *ACM Computer Surveys* (1981).

[13] BREWER, E. Towards robust distributed systems. PODC Keynote, 2000.

[14] BRONSON, N., AMSDEN, Z., CABRERA, G., CHAKKA, P., DIMOV, P., DING, H., FERRIS, J., GIARDULLO, A., KULKARNI, S., LI, H., MARCHUKOV, M., PETROV, D., PUZAR, L., SONG, Y. J., AND VENKATARAMANI, V. Tao: Facebook's distributed data store for the social graph. In *USENIX ATC* (2013).

[15] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. *ACM TOCS 26*, 2 (2008).

[16] COOPER, B. F., RAMAKRISHNAN, R., SRIVASTAVA, U., SILBERSTEIN, A., BOHANNON, P., JACOBSEN, H.-A., PUZ, N., WEAVER, D., AND YERNENI, R. PNUTS: Yahoo!'s hosted data serving platform. In *VLDB* (2008).

[17] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, 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. In *OSDI* (2012).

[18] COWLING, J., AND LISKOV, B. Granola: low-overhead distributed transaction coordination. In *USENIX ATC* (2012).

[19] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's highly available key-value store. In *SOSP* (2007).

[20] Dey, A., Fekete, A., Nambiar, R., and Rohm, U. YCSB+T: Benchmarking web-scale transactional databases. In *ICDEW* (2014).

[21] Du, J., Elnikety, S., Roy, A., and Zwaenepoel, W. Orbe: Scalable causal consistency using dependency matrices and physical clocks. In *SOCC* (2013).

[22] Du, J., Iorgulescu, C., Roy, A., and Zwaenepoel, W. Gentlerain: Cheap and scalable causal consistency with physical clocks. In *SOCC* (2014).

[23] Escriva, R., Wong, B., and Sirer, E. G. HyperKV: A distributed, searchable key-value store for cloud computing. In *SIGCOMM* (2012).

[24] Fitzpatrick, B. Memcached: a distributed memory object caching system. `http://memcached.org/`, 2014.

[25] Gifford, D. K. *Information Storage in a Decentralized Computer System*. PhD thesis, Stanford University, 1981.

[26] Gilbert, S., and Lynch, N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News* (2002).

[27] Glendenning, L., Beschastnikh, I., Krishnamurthy, A., and Anderson, T. Scalable consistency in Scatter. In *SOSP* (2011).

[28] Golab, W., Rahman, M. R., AuYoung, A., Keeton, K., and Gupta, I. Client-centric benchmarking of eventual consistency for cloud storage systems. In *ICDCS* (2014).

[29] Herlihy, M. P., and Wing, J. M. Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS* (1990).

[30] Huang, Q., Gudmundsdottir, H., Vigfusson, Y., Freedman, D. A., Birman, K., and van Renesse, R. Characterizing load imbalance in real-world networked caches. In *HotNets* (2014).

[31] Kraska, T., Pang, G., Franklin, M. J., Madden, S., and Fekete, A. MDCC: Multi-data center consistency. In *EuroSys* (2013).

[32] Lakshman, A., and Malik, P. Cassandra – a decentralized structured storage system. In *LADIS* (Oct. 2009).

[33] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Comm. ACM* (1978).

[34] Lamport, L. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computer* (1979).

[35] Lamport, L. On interprocess communication. Part I: Basic formalism and Part II: Algorithms. *Distributed Computing* (1986).

[36] Li, C., Porto, D., Clement, A., Gehrke, J., Preguiça, N., and Rodrigues, R. Making geo-replicated systems fast as possible, consistent when necessary. In *OSDI* (2012).

[37] Lipton, R. J., and Sandberg, J. S. PRAM: A scalable shared memory. Tech. Rep. TR-180-88, Princeton Univ., Dept. Comp. Sci., 1988.

[38] 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 *SOSP* (2011).

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

[40] Mills, D., Martin, J., Burbank, J., and Kasch, W. Network time protocol version 4: Protocol and algorithms specification. *Internet Engineering Task Force (IETF)* (2010).

[41] Mu, S., Cui, Y., Zhang, Y., Lloyd, W., and Li, J. Extracting more concurrency from distributed transactions. In *OSDI* (2014).

[42] NISHTALA, R., FUGAL, H., GRIMM, S., KWIATKOWSKI, M., LEE, H., LI, H. C., MCELROY, R., PALECZNY, M., PEEK, D., SAAB, P., STAFFORD, D., TUNG, T., AND VENKATARAMANI, V. Scaling memcache at Facebook. In *NSDI* (2013).

[43] OUSTERHOUT, J., AGRAWAL, P., ERICKSON, D., KOZYRAKIS, C., LEVERICH, J., MAZIÈRES, D., MITRA, S., NARAYANAN, A., ROSENBLUM, M., RUMBLE, S. M., STRATMANN, E., AND STUTSMAN, R. The case for RAMCloud. *ACM SIGOPS Operating Systems Review* (2010).

[44] PAPADIMITRIOU, C. H. The serializability of concurrent database updates. *Journal of the ACM* (1979).

[45] PATIL, S., POLTE, M., REN, K., TANTISIRIROJ, W., XIAO, L., L'OPEZ, J., GIBSON, G., FUCHS, A., AND RINALDI, B. YCSB++: benchmarking and performance debugging advanced features in scalable table stores. In *SOCC* (2011).

[46] `http://project-voldemort.com/`, 2011.

[47] RAHMAN, M. R., GOLAB, W., AUYOUNG, A., KEETON, K., AND WYLIE, J. J. Toward a principled framework for benchmarking consistency. In *HotDep* (2012).

[48] SHARMA, Y., AJOUX, P., ANG, P., CALLIES, D., CHOUDHARY, A., DEMAILLY, L., FERSCH, T., GUZ, L. A., KOTULSKI, A., KULKARNI, S., KUMAR, S., LI, H., LI, J., MAKEEV, E., PRAKASAM, K., VAN RENESSE, R., ROY, S., SETH, P., SONG, Y. J., VEERARAGHAVAN, K., WESTER, B., AND XIE, P. Wormhole: Reliable pub-sub to support geo-replicated internet services. In *NSDI* (May 2015).

[49] SHUTE, J., OANCEA, M., ELLNER, S., HANDY, B., ROLLINS, E., SAMWEL, B., VINGRALEK, R., WHIPKEY, C., CHEN, X., JEGERLEHNER, B., LITTLEFIELD, K., AND TONG, P. F1: The fault-tolerant distributed RDBMS supporting google's ad business. In *SIGMOD* (2012).

[50] SOVRAN, Y., POWER, R., AGUILERA, M. K., AND LI, J. Transactional storage for geo-replicated systems. In *SOSP* (2011).

[51] 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 *SOSP* (2013).

[52] THOMSON, A., DIAMOND, T., WENG, S.-C., REN, K., SHAO, P., AND ABADI, D. J. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD* (2012).

[53] WADA, H., FEKETE, A., ZHAO, L., LEE, K., AND LIU, A. Data consistency properties and the trade-offs in commercial cloud storage: the consumers' perspective. In *CIDR* (2011).

[54] XIE, C., SU, C., KAPRITSOS, M., WANG, Y., YAGHMAZADEH, N., ALVISI, L., AND MAHAJAN, P. Salt: combining acid and base in a distributed database. In *OSDI* (2014).

[55] ZELLAG, K., AND KEMME, B. How consistent is your cloud application? In *SOCC* (2012).

[56] ZHANG, Y., POWER, R., ZHOU, S., SOVRAN, Y., AGUILERA, M. K., AND LI, J. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *SOSP* (2013).