

# TAO: Facebook's Distributed Data Store for the Social Graph

Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov  
Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov  
Dmitri Petrov, Lovro Puzar, Yee Jiun Song, Venkat Venkataramani  
*Facebook, Inc.*

## Abstract

We introduce a simple data model and API tailored for serving the social graph, and TAO, an implementation of this model. TAO is a geographically distributed data store that provides efficient and timely access to the social graph for Facebook's demanding workload using a fixed set of queries. It is deployed at Facebook, replacing memcache for many data types that fit its model. The system runs on thousands of machines, is widely distributed, and provides access to many petabytes of data. TAO can process a billion reads and millions of writes each second.

检索 写入

## 1 Introduction

Facebook has more than a billion active users who record their relationships, share their interests, upload text, images, and video, and curate semantic information about their data [2]. The personalized experience of social applications comes from timely, efficient, and scalable access to this flood of data, the *social graph*. In this paper we introduce TAO, a read-optimized graph data store we have built to handle a demanding Facebook workload.

Before TAO, Facebook's web servers directly accessed MySQL to read or write the social graph, aggressively using memcache [21] as a lookaside cache. TAO implements a graph abstraction directly, allowing it to avoid some of the fundamental shortcomings of a lookaside cache architecture. TAO continues to use MySQL for persistent storage, but mediates access to the database and uses its own graph-aware cache.

TAO is deployed at Facebook as a single geographically distributed instance. It has a minimal API and explicitly favors availability and per-machine efficiency over strong consistency; its novelty is its scale: TAO can sustain a billion reads per second on a changing data set of many petabytes.

Overall, this paper makes three contributions. We motivate (§ 2) and characterize (§ 7) a challenging workload: efficient and available read-mostly access to a changing graph. We describe objects and associations, a data model and API that we use to access the graph (§ 3). Lastly, we detail TAO, a geographically distributed system that implements this API (§§ 4–6), and evaluate its performance on our workload (§ 8).

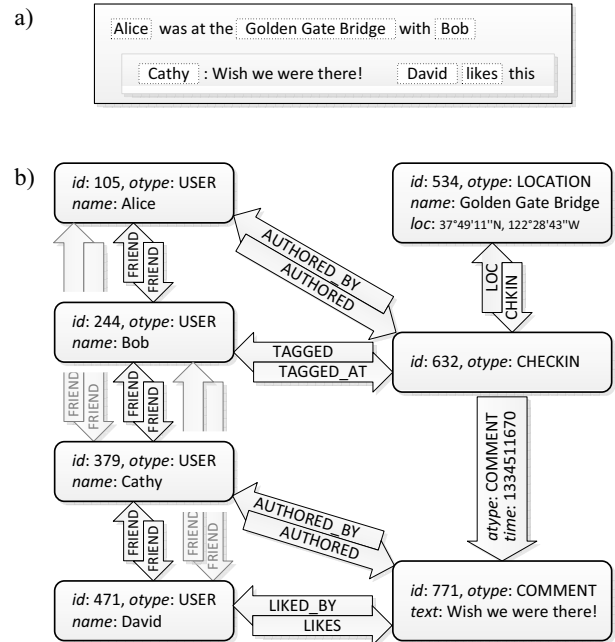


Figure 1: A running example of how a user's checkin might be mapped to objects and associations.

## 2 Background

A single Facebook page may aggregate and filter hundreds of items from the social graph. We present each user with content tailored to them, and we filter every item with privacy checks that take into account the current viewer. This extreme customization makes it infeasible to perform most aggregation and filtering when content is created; instead we resolve data dependencies and check privacy each time the content is viewed. As much as possible we pull the social graph, rather than pushing it. This implementation strategy places extreme read demands on the graph data store; it must be efficient, highly available, and scale to high query rates.

### 2.1 Serving the Graph from Memcache

Facebook was originally built by storing the social graph in MySQL, querying it from PHP, and caching results in memcache [21]. This lookaside cache architecture is well suited to Facebook's rapid iteration cycles, since all

of the data mapping and cache-invalidation computations are in client code that is deployed frequently. Over time a PHP abstraction was developed that allowed developers to read and write the **objects (nodes)** and **associations (edges)** in the graph, and direct access to MySQL was deprecated for data types that fit the model.

TAO is a service we constructed that **directly implements the objects and associations model**. We were motivated by encapsulation failures in the PHP API, by the opportunity to access the graph easily from non-PHP services, and by several fundamental problems with the lookaside cache architecture:

**Inefficient edge lists:** A key-value cache is not a good semantic fit for lists of edges; queries must always fetch the entire edge list and changes to a single edge require the entire list to be reloaded. Basic list support in a lookaside cache would only address the first problem; something much more complicated is required to coordinate concurrent incremental updates to cached lists.

**Distributed control logic:** In a lookaside cache architecture the control logic is run on clients that don't communicate with each other. This increases the number of failure modes, and makes it difficult to avoid thundering herds. Nishtala et al. provide an in-depth discussion of the problems and present *leases*, a general solution [21]. For objects and associations the fixed API allows us to move the control logic into the cache itself, where the problem can be solved more efficiently.

**Expensive read-after-write consistency:** Facebook uses asynchronous master/slave replication for MySQL, which poses a problem for caches in data centers using a replica. Writes are forwarded to the master, but some time will elapse before they are reflected in the local replica. Nishtala et al.'s *remote markers* [21] track keys that are known to be stale, forwarding reads for those keys to the master region. By restricting the data model to objects and associations we can update the replica's cache at write time, then use graph semantics to interpret cache maintenance messages from concurrent updates. This provides (in the absence of multiple failures) read-after-write consistency for all clients that share a cache, without requiring inter-regional communication.

## 2.2 TAO's Goal


TAO provides **basic access to the nodes and edges** of a constantly changing graph in data centers across multiple regions. It is optimized heavily for reads, and explicitly favors efficiency and availability over consistency.

A system like TAO is likely to be useful for any application domain that needs to **efficiently generate fine-grained customized content** from highly interconnected data. The application should not expect the data to be stale in the common case, but should be able to tolerate it. Many social networks fit in this category.

## 3 TAO Data Model and API

Facebook focuses on people, actions, and relationships. We model these entities and connections as nodes and edges in a graph. This representation is very flexible; it directly models real-life objects, and can also be used to store an application's internal implementation-specific data. TAO's goal is **not to support a complete set of graph queries, but to provide sufficient expressiveness to handle most application needs while allowing a scalable and efficient implementation**.

Consider the social networking example in Figure 1a, in which Alice used her mobile phone to record her visit to a famous landmark with Bob. She 'checked in' to the Golden Gate Bridge and 'tagged' Bob to indicate that he is with her. Cathy added a comment that David has 'liked.' The social graph includes the users (Alice, Bob, Cathy, and David), their relationships, their actions (checking in, commenting, and liking), and a physical location (the Golden Gate Bridge).

Facebook's application servers would query this event's underlying nodes and edges every time it is rendered. Fine-grained privacy controls mean that **each user may see a different view of the checkin**; the individual nodes and edges that encode the activity can be reused for all of these views, but the aggregated content and the results of privacy checks cannot. 

### 3.1 Objects and Associations

TAO *objects* are **typed nodes**, and TAO *associations* are **typed directed edges between objects**. Objects are identified by a 64-bit integer (id) that is unique across all objects, regardless of object type (otype). Associations are identified by the source object (id1), association type (atype) and destination object (id2). At most one association of a given type can exist between any two objects. Both objects and associations may contain data as key→value pairs. A per-type schema lists the possible keys, the value type, and a default value. Each association has a 32-bit time field, which plays a central role in queries<sup>1</sup>.

**Object:** (id) → (otype, (key → value)\*)

**Assoc.:** (id1, atype, id2) → (time, (key → value)\*)

Figure 1b shows how TAO objects and associations might encode the example, with some data and times omitted for clarity. The example's users are represented by objects, as are the checkin, the landmark, and Cathy's comment. Associations capture the users' friendships, authorship of the checkin and comment, and the binding between the checkin and its location and comments.

<sup>1</sup>The time field is actually a generic application-assigned integer.

Actions may be encoded either as objects or associations. Both Cathy's comment and David's 'like' represent actions taken by a user, but only the comment results in a new object. Associations naturally model actions that can happen at most once or record state transitions, such as the acceptance of an event invitation, while repeatable actions are better represented as objects.

Although associations are directed, it is common for an association to be tightly coupled with an inverse edge. In this example all of the associations have an inverse except for the link of type COMMENT. No inverse edge is required here since the application does not traverse from the comment to the CHECKIN object. Once the checkin's id is known, rendering Figure 1a only requires traversing outbound associations. Discovering the checkin object, however, requires the inbound edges or that an id is stored in another Facebook system.

The schemas for object and association types describe only the data contained in instances. They do not impose any restrictions on the edge types that can connect to a particular node type, or the node types that can terminate an edge type. The same atype is used to represent authorship of the checkin object and the comment object in Figure 1, for example. Self-edges are allowed.

## 3.2 Object API

TAO's object API provides operations to allocate a new object and id, and to retrieve, update, or delete the object associated with an id. A notable omission is a compare-and-set functionality, whose usefulness is substantially reduced by TAO's eventual consistency semantics. The update operation can be applied to a subset of the fields.

## 3.3 Association API

Many edges in the social graph are bidirectional, either symmetrically like the example's FRIEND relationship or asymmetrically like AUTHORED and AUTHORED\_BY. Bidirectional edges are modeled as two separate associations. TAO provides support for keeping associations in sync with their inverses, by allowing association types to be configured with an inverse type. For such associations, creations, updates, and deletions are automatically coupled with an operation on the inverse association. Symmetric bidirectional types are their own inverses. The association write operations are:

- **assoc\_add(id1, atype, id2, time, (k→v)\*)** – Adds or overwrites the association (id1, atype, id2), and its inverse (id1, inv(atype), id2) if defined.
- **assoc\_delete(id1, atype, id2)** – Deletes the association (id1, atype, id2) and the inverse if it exists.
- **assoc\_change\_type(id1, atype, id2, newtype)** – Changes the association (id1, atype, id2) to (id1, newtype, id2), if (id1, atype, id2) exists.

## 3.4 Association Query API

The starting point for any TAO association query is an originating object and an association type. This is the natural result of searching for a specific type of information about a particular object. Consider the example in Figure 1. In order to display the CHECKIN object, the application needs to enumerate all tagged users and the most recently added comments.

A characteristic of the social graph is that most of the data is old, but many of the queries are for the newest subset. This creation-time locality arises whenever an application focuses on recent items. If the Alice in Figure 1 is a famous celebrity then there might be thousands of comments attached to her checkin, but only the most recent ones will be rendered by default.

TAO's association queries are organized around *association lists*. We define an association list to be the list of all associations with a particular id1 and atype, arranged in descending order by the time field:

**Association List:**  $(id1, atype) \rightarrow [a_{new} \dots a_{old}]$

For example, the list  $(i, COMMENT)$  has edges to the example's comments about  $i$ , most recent first.

TAO's queries on associations lists:

- **assoc\_get(id1, atype, id2set, high?, low?)** – returns all of the associations (id1, atype, id2) and their time and data, where  $id2 \in id2set$  and  $high \geq time \geq low$  (if specified). The optional time bounds are to improve cacheability for large association lists (see § 5).
- **assoc\_count(id1, atype)** – returns the size of the association list for (id1, atype), which is the number of edges of type atype that originate at id1.
- **assoc\_range(id1, atype, pos, limit)** – returns elements of the (id1, atype) association list with index  $i \in [pos, pos + limit)$ .
- **assoc\_time\_range(id1, atype, high, low, limit)** – returns elements from the (id1, atype) association list, starting with the first association where  $time \leq high$ , returning only edges where  $time \geq low$ .

TAO enforces a per-atype upper bound (typically 6,000) on the actual limit used for an association query. To enumerate the elements of a longer association list the client must issue multiple queries, using pos or high to specify a starting point.

For the example shown in Figure 1 we can map some possible queries to the TAO API as follows:

- “50 most recent comments on Alice's checkin”  $\Rightarrow$  `assoc_range(632, COMMENT, 0, 50)`
- “How many checkins at the GG Bridge?”  $\Rightarrow$  `assoc_count(534, CHECKIN)`



## 4 TAO Architecture

In this section we describe the units that make up TAO, and the multiple layers of aggregation that allow it to scale across data centers and geographic regions. TAO is separated into two caching layers and a storage layer.

### 4.1 Storage Layer

Objects and associations were stored in MySQL at Facebook even before TAO was built; it was the backing store for the original PHP implementation of the API. This made it the natural choice for TAO's persistent storage.

The TAO API is mapped to a small set of simple SQL queries, but it could also be mapped efficiently to range scans in a non-SQL data storage system such as LevelDB [3] by explicitly maintaining the required indexes. When evaluating the suitability of a backing store for TAO, however, it is important to consider the data accesses that don't use the API. These include backups, bulk import and deletion of data, bulk migrations from one data format to another, replica creation, asynchronous replication, consistency monitoring tools, and operational debugging. An alternate store would also have to provide atomic write transactions, efficient granular writes, and few latency outliers.

Given that TAO needs to handle a far larger volume of data than can be stored on a single MySQL server, we divide data into logical *shards*. Each shard is contained in a logical database. Database servers are responsible for one or more shards. In practice, the number of shards far exceeds the number of servers; we tune the shard to server mapping to balance load across different hosts. By default all object types are stored in one table, and all association types in another.

Each object id contains an embedded `shard_id` that identifies its hosting shard. Objects are bound to a shard for their entire lifetime. An association is stored on the shard of its `id1`, so that every association query can be served from a single server. Two ids are unlikely to map to the same server unless they were explicitly colocated at creation time.

### 4.2 Caching Layer

TAO's cache implements the complete API for clients, handling all communication with databases. The caching layer consists of multiple *cache servers* that together form a *tier*. A tier is collectively capable of responding to any TAO request. (We also refer to the set of databases in one region as a tier.) Each request maps to a single cache server using a sharding scheme similar to the one described in § 4.1. There is no requirement that tiers have the same number of hosts.

Clients issue requests directly to the appropriate cache server, which is then responsible for completing the read

or write. For cache misses and write requests, the server contacts other caches and/or databases.

The TAO in-memory cache contains objects, association lists, and association counts. We fill the cache on demand and evict items using a least recently used (LRU) policy. Cache servers understand the semantics of their contents and use them to answer queries even if the exact query has not been previously processed, e.g. a cached count of zero is sufficient to answer a range query.

Write operations on an association with an inverse may involve two shards, since the forward edge is stored on the shard for `id1` and the inverse edge is on the shard for `id2`. The tier member that receives the query from the client issues an RPC call to the member hosting `id2`, which will contact the database to create the inverse association. Once the inverse write is complete, the caching server issues a write to the database for `id1`. TAO does not provide atomicity between the two updates. If a failure occurs the forward may exist without an inverse; these *hanging* associations are scheduled for repair by an asynchronous job.

### 4.3 Client Communication Stack

It is common for hundreds of objects and associations to be queried while rendering a Facebook page, which is likely to require communication with many cache servers in a short period of time. The challenges of the resulting all-to-all communication are similar to those faced by our memcache pools. TAO and memcache share most of the client stack described by Nishtala et al. [21]. The latency of TAO requests can be much higher than those of memcache, because TAO requests may access the database, so to avoid head-of-line blocking on multiplexed connections we use a protocol with out-of-order responses.

### 4.4 Leaders and Followers

In theory a single cache tier could be scaled to handle any foreseeable aggregate request rate, so long as shards are small enough. In practice, though, large tiers are problematic because they are more prone to hot spots and they have a quadratic growth in all-to-all connections.

To add servers while limiting the maximum tier size we split the cache into two levels: a *leader* tier and multiple *follower* tiers. Some of TAO's advantages over a lookaside cache architecture (as described in § 2.1) rely on having a single cache coordinator per database; this split allows us to keep the coordinators in a single tier per region. As in the single-tier configuration, each tier contains a set of cache servers that together are capable of responding to any TAO query; that is, every shard in the system maps to one caching server in each tier. Leaders (members of the leader tier) behave as described in § 4.2, reading from and writing to the storage layer. Fol-

lowers (members of follower tiers) will instead forward read misses and writes to a leader. Clients communicate with the closest follower tier and never contact leaders directly; if the closest follower is unavailable they fail over to another nearby follower tier.

Given this two-level caching hierarchy, care must be taken to keep TAO caches consistent. Each shard is hosted by one leader, and all writes to the shard go through that leader, so it is naturally consistent. Followers, on the other hand, must be explicitly notified of updates made via other follower tiers.

TAO provides eventual consistency [33, 35] by asynchronously sending cache maintenance messages from the leader to the followers. An object update in the leader enqueues invalidation messages to each corresponding follower. The follower that issued the write is updated synchronously on reply from the leader; a version number in the cache maintenance message allows it to be ignored when it arrives later. Since we cache only contiguous prefixes of association lists, invalidating an association might truncate the list and discard many edges. Instead, the leader sends a *refill* message to notify followers about an association write. If a follower has cached the association, then the refill request triggers a query to the leader to update the follower's now-stale association list. § 6.1 discusses the consistency of this design and also how it tolerates failures.

Leaders serialize concurrent writes that arrive from followers. Because a single leader mediates all of the requests for an id1, it is also ideally positioned to protect the database from thundering herds. The leader ensures that it does not issue concurrent overlapping queries to the database and also enforces a limit on the maximum number of pending queries to a shard.

## 4.5 Scaling Geographically

The leader and followers configuration allows TAO to scale to handle a high workload, since read throughput scales with the total number of follower servers in all tiers. Implicit in the design, however, is the assumption that the network latencies from follower to leader and leader to database are low. This assumption is reasonable if clients are restricted to a single data center, or even to a set of data centers in close proximity. It is not true, however, in our production environment.

As our social networking application's computing and network requirements have grown, we have had to expand beyond a single geographical location: today, follower tiers can be thousands of miles apart. In this configuration, network round trip times can quickly become the bottleneck of the overall architecture. Since read misses by followers are 25 times as frequent as writes in our workloads, we chose a master/slave architecture that requires writes to be sent to the master, but that allows

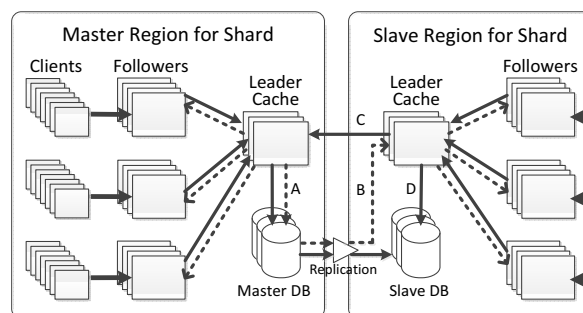


Figure 2: Multi-region TAO configuration. The master region sends read misses, writes, and embedded consistency messages to the master database (A). Consistency messages are delivered to the slave leader (B) as the replication stream updates the slave database. Slave leader sends writes to the master leader (C) and read misses to the replica DB (D). The choice of master and slave is made separately for each shard.

read misses to be serviced locally. As with the leader/follower design, we propagate update notifications asynchronously to maximize performance and availability, at the expense of data freshness.

The social graph is tightly interconnected; it is not possible to group users so that cross-partition requests are rare. This means that each TAO follower must be local to a tier of databases holding a complete multi-petabyte copy of the social graph. It would be prohibitively expensive to provide full replicas in every data center.

Our solution to this problem is to choose data center locations that are clustered into only a few *regions*, where the intra-region latency is small (typically less than 1 millisecond). It is then sufficient to store one complete copy of the social graph per region. Figure 2 shows the overall architecture of the master/slave TAO system.

Followers behave identically in all regions, forwarding read misses and writes to the local region's leader tier. Leaders query the local region's database regardless of whether it is the master or slave. Writes, however, are forwarded by the local leader to the leader that is in the region with the master database. This means that read latency is independent of inter-region latency.

The master region is controlled separately for each shard, and is automatically switched to recover from the failure of a database. Writes that fail during the switch are reported to the client as failed, and are not retried. Note that since each cache hosts multiple shards, a server may be both a master and a slave at the same time. We prefer to locate all of the master databases in a single region. When an inverse association is mastered in a different region, TAO must traverse an extra inter-region link to forward the inverse write.

TAO embeds invalidation and refill messages in the database replication stream. These messages are delivered in a region immediately after a transaction has been replicated to a slave database. Delivering such messages earlier would create cache inconsistencies, as reading from the local database would provide stale data. At Facebook TAO and memcache use the same pipeline for delivery of invalidations and refills [21].

If a forwarded write is successful then the local leader will update its cache with the fresh value, even though the local slave database probably has not yet been updated by the asynchronous replication stream. In this case followers will receive two invalidates or refills from the write, one that is sent when the write succeeds and one that is sent when the write's transaction is replicated to the local slave database.

TAO's master/slave design ensures that all reads can be satisfied within a single region, at the expense of potentially returning stale data to clients. As long as a user consistently queries the same follower tier, the user will typically have a consistent view of TAO state. We discuss exceptions to this in the next section.

## 5 Implementation

Previous sections describe how TAO servers are aggregated to handle large volumes of data and query rates. This section details important optimizations for performance and storage efficiency.

### 5.1 Caching Servers

TAO's caching layer serves as an intermediary between clients and the databases. It aggressively caches objects and associations to provide good read performance.

TAO's memory management is based on Facebook's customized memcached, as described by Nishtala et al. [21]. TAO has a slab allocator that manages slabs of equal size items, a thread-safe hash table, LRU eviction among items of equal size, and a dynamic slab rebalancer that keeps the LRU eviction ages similar across all types of slabs. A slab item can hold one node or one edge list.

To provide better isolation, TAO partitions the available RAM into *arenas*, selecting the arena by the object or association type. This allows us to extend the cache lifetime of important types, or to prevent poor cache citizens from evicting the data of better-behaved types. So far we have only manually configured arenas to address specific problems, but it should be possible to automatically size arenas to improve TAO's overall hit rate.

For small fixed-size items, such as association counts, the memory overhead of the pointers for bucket items in the main hash table becomes significant. We store these items separately, using direct-mapped 8-way associative caches that require no pointers. LRU order within each

bucket is tracked by simply sliding the entries down. We achieve additional memory efficiency by adding a table that maps the each active atype to a 16 bit value. This lets us map (id1, atype) to a 32-bit count in 14 bytes; a negative entry, which records the absence of any id2 for an (id1, atype), takes only 10 bytes. This optimization allows us to hold about 20% more items in cache for a given system configuration.

### 5.2 MySQL Mapping

Recall that we divide the space of objects and associations into shards. Each shard is assigned to a logical MySQL database that has a table for objects and a table for associations. All of the fields of an object are serialized into a single 'data' column. This approach allows us to store objects of different types within the same table. Objects that benefit from separate data management policies are stored in separate custom tables.

Associations are stored similarly to objects, but to support range queries, their tables have an additional index based on id1, atype, and time. To avoid potentially expensive SELECT COUNT queries, association counts are stored in a separate table.

### 5.3 Cache Sharding and Hot Spots

Shards are mapped onto cache servers within a tier using consistent hashing [15]. This simplifies tier expansions and request routing. However, this semi-random assignment of shards to cache servers can lead to load imbalance: some followers will shoulder a larger portion of the request load than others. TAO rebalances load among followers with *shard cloning*, in which reads to a shard are served by multiple followers in a tier. Consistency management messages for a cloned shard are sent to all followers hosting that shard.

In our workloads, it is not uncommon for a popular object to be queried orders of magnitude more often than other objects. Cloning can distribute this load across many followers, but the high hit rate for these objects makes it worthwhile to place them in a small client-side cache. When a follower responds to a query for a hot item, it includes the object or association's access rate. If the access rate exceeds a certain threshold, the TAO client caches the data and version. By including the version number in subsequent queries, the follower can omit the data in replies if the data has not changed since the previous version. The access rate can also be used to throttle client requests for very hot objects.

### 5.4 High-Degree Objects

Many objects have more than 6,000 associations with the same atype emanating from them, so TAO does not cache

the complete association list. It is also common that `assoc_get` queries are performed that have an empty result (no edge exists between the specified `id1` and `id2`). Unfortunately, for high-degree objects these queries will always go to the database, because the queried `id2` could be in the uncached tail of the association list.

We have addressed this inefficiency in the cache implementation by modifying client code that is observed to issue problematic queries. One solution to this problem is to use `assoc_count` to choose the query direction, since checking for the inverse edge is equivalent. In some cases where both ends of an edge are high-degree nodes, we can also leverage application-domain knowledge to improve cacheability. Many associations set the time field to their creation time, and many objects include their creation time as a field. Since an edge to a node can only be created after the node has been created, we can limit the `id2` search to associations whose time is  $\geq$  than the object's creation time. So long as an edge older than the object is present in cache then this query can be answered directly by a TAO follower.

## 6 Consistency and Fault Tolerance

Two of the most important requirements for TAO are availability and performance. When failures occur we would like to continue to render Facebook, even if the data is stale. In this section, we describe the consistency model of TAO under normal operation, and how TAO sacrifices consistency under failure modes.

### 6.1 Consistency

Under normal operation, objects and associations in TAO are eventually consistent [33, 35]; after a write, TAO guarantees the eventual delivery of an invalidation or refill to all tiers. Given a sufficient period of time during which external inputs have quiesced, all copies of data in TAO will be consistent and reflect all successful write operations to all objects and associations. Replication lag is usually less than one second.

In normal operation (at most one failure encountered by a request) TAO provides read-after-write consistency within a single tier. TAO synchronously updates the cache with locally written values by having the master leader return a *changeset* when the write is successful. This changeset is propagated through the slave leader (if any) to the follower tier that originated the write query. If an inverse type is configured for an association, then writes to associations of that type may affect both the `id1`'s and the `id2`'s shard. In these cases, the changeset returned by the master leader contains both updates, and the slave leader (if any) and the follower that forwarded the write must each send the changeset to the `id2`'s shard in their respective tiers, before returning to the caller.

The changeset cannot always be safely applied to the follower's cache contents, because the follower's cache may be stale if the refill or invalidate from a second follower's update has not yet been delivered. We resolve this race condition in most cases with a version number that is present in the persistent store and the cache. The version number is incremented during each update, so the follower can safely invalidate its local copy of the data if the changeset indicates that its pre-update value was stale. Version numbers are not exposed to the TAO clients. In slave regions, this scheme is vulnerable to a rare race condition between cache eviction and storage server update propagation. The slave storage server may hold an older version of a piece of data than what is cached by the caching server, so if the post-changeset entry is evicted from cache and then reloaded from the database, a client may observe a value go back in time in a single follower tier. Such a situation can only occur if it takes longer for the slave region's storage server to receive an update than it does for a cached item to be evicted from cache, which is rare in practice.

Although TAO does not provide strong consistency for its clients, because it writes to MySQL synchronously the master database is a consistent source of truth. This allows us to provide stronger consistency for the small subset of requests that need it. TAO reads may be marked *critical*, in which case they will be proxied to the master region. We could use critical reads during an authentication process, for example, so that replication lag doesn't allow use of stale credentials.

### 6.2 Failure Detection and Handling

TAO scales to thousands of machines over multiple geographical locations, so transient and permanent failures are commonplace. Therefore, it is important that TAO detect potential failures and route around them. TAO servers employ aggressive network timeouts so as not to continue waiting on responses that may never arrive. Each TAO server maintains per-destination timeouts, marking hosts as down if there are several consecutive timeouts, and remembering downed hosts so that subsequent requests can be proactively aborted. This simple failure detector works well, although it does not always preserve full capacity in a brown-out scenario, such as bursty packet drops that limit TCP throughput. Upon detection of a failed server, TAO routes around the failures in a best effort fashion in order to preserve availability and performance at the cost of consistency. We actively probe failed machines to discover when (if) they recover.

**Database failures:** Databases are marked down in a global configuration if they crash, if they are taken offline for maintenance, or if they are replicating from a master database and they get too far behind. When a



master database is down, one of its slaves is automatically promoted to be the new master.

When a region's slave database is down, cache misses are redirected to the TAO leaders in the region hosting the database master. Since cache consistency messages are embedded in the database's replication stream, however, they can't be delivered by the primary mechanism. During the time that a slave database is down an additional binlog tailer is run on the master database, and the refills and invalidations are delivered inter-regionally. When the slave database comes back up, invalidation and refill messages from the outage will be delivered again.

**Leader failures:** When a leader cache server fails, followers automatically route read and write requests around it. Followers reroute read misses directly to the database. Writes to a failed leader, in contrast, are rerouted to a random member of the leader's tier. This replacement leader performs the write and associated actions, such as modifying the inverse association and sending invalidations to followers. The replacement leader also enqueues an asynchronous invalidation to the original leader that will restore its consistency. These asynchronous invalidates are recorded both on the coordinating node and inserted into the replication stream, where they are spooled until the leader becomes available. If the failing leader is partially available then followers may see a stale value until the leader's consistency is restored.

**Refill and invalidation failures:** Leaders send refills and invalidations asynchronously. If a follower is unreachable, the leader queues the message to disk to be delivered at a later time. Note that a follower may be left with stale data if these messages are lost due to permanent leader failure. This problem is solved by a bulk invalidation operation that invalidates all objects and associations from a `shard_id`. After a failed leader box is replaced, all of the shards that map to it must be invalidated in the followers, to restore consistency.

**Follower failures:** In the event that a TAO follower fails, followers in other tiers share the responsibility of serving the failed host's shards. We configure each TAO client with a primary and backup follower tier. In normal operations requests are sent only to the primary. If the server that hosts the shard for a particular request has been marked down due to timeouts, then the request is sent instead to that shard's server in the backup tier. Because failover requests still go to a server that hosts the corresponding shard, they are fully cacheable and do not require extra consistency work. Read and write requests from the client are failed over in the same way. Note that failing over between different tiers may cause read-after-write consistency to be violated if the read reaches the failover target before the write's refill or invalidate.

read requests	99.8 %	write requests	0.2 %
assoc_get	15.7 %	assoc_add	52.5 %
assoc_range	40.9 %	assoc_del	8.3 %
assoc_time_range	2.8 %	assoc_change_type	0.9 %
assoc_count	11.7 %	obj_add	16.5 %
obj_get	28.9 %	obj_update	20.7 %
		obj_delete	2.0 %

Figure 3: Relative frequencies for client requests to TAO from all Facebook products. Reads account for almost all of the calls to the API.

## 7 Production Workload

Facebook has a single instance of TAO in production. Multi-tenancy in a system such as TAO allows us to amortize operational costs and share excess capacity among clients. It is also an important enabler for rapid product innovation, because new applications can link to existing data and there is no need to move data or provision servers as an application grows from one user to hundreds of millions. Multi-tenancy is especially important for objects, because it allows the entire 64-bit id space to be handled uniformly without an extra step to resolve the otype.

The TAO system contains many follower tiers spread across several geographic regions. Each region has one complete set of databases, one leader cache tier, and at least two follower tiers. Our TAO deployment continuously processes a billion reads and millions of writes per second. We are not aware of another geographically distributed graph data store at this scale.

To characterize the workload that is seen by TAO, we captured a random sample of 6.5 million requests over a 40 day period. In this section, we describe the results of an analysis of that sample.

At a high level, our workload shows the following characteristics:

- reads are much more frequent than writes;
- most edge queries have empty results; and
- query frequency, node connectivity, and data size have distributions with long tails.

Figure 3 breaks down the load on TAO. Reads dominate, with only 0.2% of requests involving a write. The majority of association reads resulted in empty association lists. Calls to `assoc_get` found an association only 19.6% of the time, 31.0% of the calls to `assoc_range` in our trace had a non-empty result, and only 1.9% of the calls to `assoc_time_range` returned any edges.

Figure 4 shows the distribution of the return values from `assoc_count`. 45% of calls return zero. Among the non-zero values, although small values are the most common, 1% of the return values were  $> 500,000$ .

Figure 5 shows the distribution of the number of asso-





Figure 4: `assoc_count` frequency in our production environment. 1% of returned counts were  $\geq 512K$ .

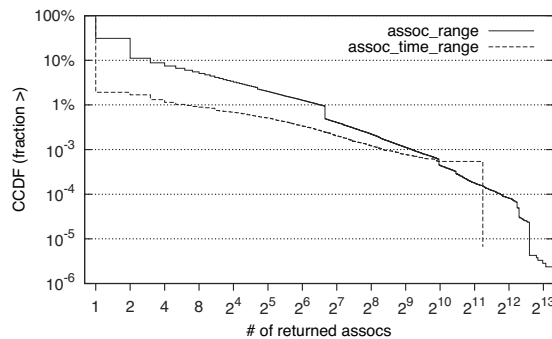


Figure 5: The number of edges returned by `assoc_range` and `assoc_time_range` queries. 64% of the non-empty results had 1 edge, 13% of which had a limit of 1.

ciations returned for range and time-range queries, and the subset that hit the limit for returned associations. Most range and time range queries had large client-supplied limits. 12% of the queries had limit = 1, but 95% of the remaining queries had limit  $\geq 1000$ . Less than 1% of the return values for queries with a limit  $\geq 1$  actually reached the limit.

Although queries for non-existent associations were common, this is not the case for objects. A valid id is only produced during object creation, so `obj.get` can only return an empty result if the object has been removed or if the object's creation has not yet been replicated to the current region. Neither of these cases occurred in our trace; every object read was successful. This doesn't mean that objects were never deleted – it just means that there was never an attempt to read a deleted object.

Figure 6 shows the distribution of the data sizes for TAO query results. 39.5% of the associations queried by clients contained no data. Our implementation allows objects to store 1MB of data and associations to store 64K of data (although a custom table must be configured for associations that store more than 255 bytes of data). The actual size of most objects and associations is much

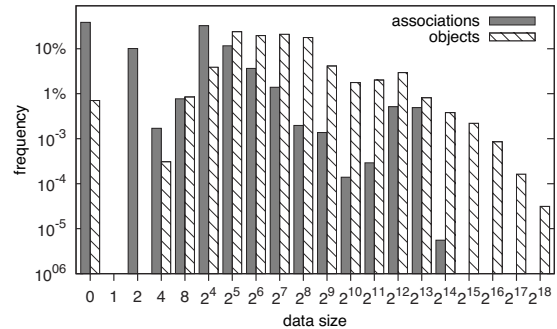


Figure 6: The size of the data stored in associations and objects that were returned by the TAO API. Associations typically store much less data than objects. The average association data size was 97.8 bytes for the 60.5% of returned associations that had some data. The average object data size was 673 bytes.

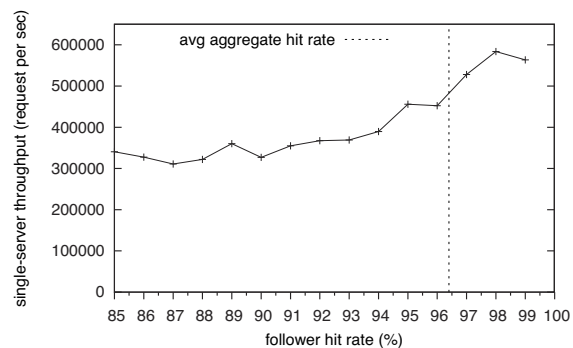


Figure 7: Throughput of an individual follower in our production environment. Cache misses and writes are more expensive than cache hits, so the peak query rate rises with hit rate. Writes are included in this graph as non-hit requests.

smaller. However, large values are frequent enough that the system must deal with them efficiently.

## 8 Performance

Running a single TAO deployment for all of Facebook allows us to benefit from economies of scale, and makes it easy for new products to integrate with existing portions of the social graph. In this section, we report on the performance of TAO under a real workload.

**Availability:** Over a period of 90 days, the fraction of failed TAO queries as measured from the web server was  $4.9 \times 10^{-6}$ . Care must be taken when interpreting this number, since the failure of one TAO query might prevent the client from issuing another query with a dynamic data dependence on the first. TAO's failures may also be correlated with those of other dependent systems.

operation	hit lat. (msec)			miss lat. (msec)		
	50%	avg	99%	50%	avg	99%
assoc_count	1.1	2.5	28.9	5.0	26.2	186.8
assoc_get	1.0	2.4	25.9	5.8	14.5	143.1
assoc_range	1.1	2.3	24.8	5.4	11.2	93.6
assoc_time_range	1.3	3.2	32.8	5.8	11.9	47.2
obj_get	1.0	2.4	27.0	8.2	75.3	186.4

Figure 8: Client-observed TAO latency in milliseconds for read requests, including client API overheads and network traversal, separated by cache hits and cache misses.

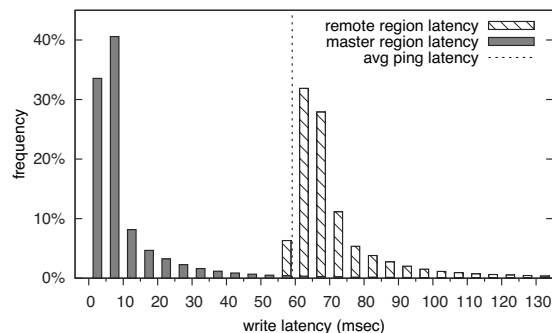


Figure 9: Write latency from clients in the same region as database masters, and from a region 58 msec away.

**Follower capacity:** The peak throughput of a follower depends on its hit rate. Figure 7 shows the highest 15-minute average throughput we observe in production for our current hardware configuration, which has 144GB of RAM, 2 Intel Xeon 8 core E5-2660 CPUs running at 2.2Ghz with Hyperthreading, and 10 Gigabit ethernet.

**Hit rates and latency:** As part of the data collection process that was described in § 7, we measured latencies in the client application; these measurements include all network latencies and the time taken to traverse the PHP TAO client stack. Requests were sampled at the same rate in all regions. TAO’s overall hit rate for reads was 96.4%. Figure 8 shows the client-observed latencies for reads. `obj_get` has higher miss latencies than the other reads because objects typically have more data (see Figure 6). `assoc_count` requests to the persistent store have a larger id1 working set than other association queries, and hence make poorer use of the database’s buffer cache.

TAO’s writes are performed synchronously to the master database, so writes from other regions include an inter-region round trip. Figure 9 compares the latency in two data centers that are 58.1 milliseconds away from each other (average round trip). Average write latency in the same region as the master was 12.1 msec; in the remote region it was  $74.4 = 58.1 + 16.3$  msec.

**Replication lag:** TAO’s asynchronous replication of writes between regions is a design trade-off that favors

read performance and throughput over consistency. We observed that TAO’s slave storage servers lag their master by less than 1 second during 85% of the tracing window, by less than 3 seconds 99% of the time, and by less than 10 seconds 99.8% of the time.

**Failover:** Follower caches directly contact the database when a leader is unavailable; this failover path was used on 0.15% of follower cache misses over our sample. Failover for write requests involves delegating those requests to a random leader, which occurred for 0.045% of association and object writes. Slave databases were promoted to be the master 0.25% of the time due to planned maintenance or unplanned downtime.

## 9 Related Work

TAO is a geographically distributed eventually consistent graph store optimized for reads. Previous distributed systems works have explored relaxed consistency, graph databases, and read-optimized storage. To our knowledge, TAO is the first to combine all of these techniques in a single system at large scale.

**Eventual consistency:** Terry et al. [33] describe eventual consistency, the relaxed consistency model which is used by TAO. Werner describes read-after-write consistency as a property of some variants of eventual consistency [35].

**Geographically distributed data stores:** The Coda file system uses data replication to improve performance and availability in the face of slow or unreliable networks [29]. Unlike Coda, TAO does not allow writes in portions of the system that are disconnected.

Megastore is a storage system that uses Paxos across geographically distributed data centers to provide strong consistency guarantees and high availability [5]. Spanner, the next generation globally distributed database developed at Google after Megastore, introduces the concept of a time API that exposes time uncertainty and leverages that to improve commit throughput and provide snapshot isolation for reads [8]. TAO addresses a very different use case, providing no consistency guarantees but handling many orders of magnitude more requests.

**Distributed hash tables and key-value systems:** Unstructured key-value systems are an attractive approach to scaling distributed storage because data can be easily partitioned and little communication is needed between partitions. Amazon’s Dynamo [10] demonstrates how they can be used in building flexible and robust commercial systems. Drawing inspiration from Dynamo, LinkedIn’s Voldemort [4] also implements a distributed key-value store but for a social network. TAO accepts lower write availability than Dynamo in exchange for avoiding the programming complexities that arise from multi-master conflict resolution. The simplicity of key-

value stores also allows for aggressive performance optimizations, as seen in Facebook’s use of memcache [21].

Many contributions in distributed hash tables have focused on routing [28, 32, 25, 24]. Li et al. [16] characterize the performance of DHTs under churn while Dabek et al. [9] focus on designing DHTs in a wide-area network. TAO exploits the hierarchy of inter-cluster latencies afforded by our data center placement and assumes a controlled environment that has few membership or cluster topology changes.

Many other works have focused on the consistency semantics provided by key-value stores. Gribble et al. [13] provide a coherent view of cached data by leveraging two-phase commit. Glendenning et al. [12] built a linearizable key-value store tolerant of churn. Sovran et al. [31] implement geo-replicated transactions.

The COPS system [17] provides causal consistency in a highly available key-value store by tracking all dependencies for all keys accessed by a client context. Eiger [18] improves on COPS by tracking conflicts between pending operations in a column-family database. The techniques used in Eiger may be applicable TAO if the per-machine efficiency can be improved.

**Hierarchical connectivity:** Nygren et al. [22] describe how the Akamai content cache optimizes latency by grouping edge clusters into regional groups that share a more powerful ‘parent’ cluster, which are similar to TAO’s follower and leader tiers.

**Structured storage:** TAO follows the recent trend of shifting away from relational databases towards structured storage approaches. While loosely defined, these systems typically provide weaker guarantees than the traditional ACID properties. Google’s BigTable [6], Yahoo!’s PNUTS [7], Amazon’s SimpleDB [1], and Apache’s HBase [34] are examples of this more scalable approach. These systems all provide consistency and transactions at the per-record or row level similar to TAO’s semantics for objects and associations, but do not provide TAO’s read efficiency or graph semantics. Escriva et al. [27] describe a *searchable* key-value store. Redis [26] is an in-memory storage system providing a range of data types and an expressive API for data sets that fit entirely in memory.

**Graph serving:** Since TAO was designed specifically to serve the social graph, it is unsurprising that it shares features with existing works on graph databases. Shao and Wang’s Trinity effort [30] stores its graph structures in-memory. Neo4j [20] is a popular open-source graph database that provides ACID semantics and the ability to shard data across several machines. Twitter uses its FlockDB [11] to store parts of its social graph, as well. To the best of our knowledge, none of these systems scale to support Facebook’s workload.

Redis [26] is a key-value store with a rich selection of

value types sufficient to efficiently implement the objects and associations API. Unlike TAO, however, it requires that the data set fit entirely in memory. Redis replicas are read-only, so they don’t provide read-after-write consistency without a higher-level system like Nishtala et al.’s remote markers [21].

**Graph processing:** TAO does not currently support an advanced graph processing API. There are several systems that try to support such operations but they are not designed to receive workloads directly from client applications. PEGASUS [14] and Yahoo’s Pig Latin [23] are systems to do data mining and analysis of graphs on top of Hadoop, with PEGASUS being focused on petascale graphs and Pig Latin focusing on a more-expressive query language. Similarly, Google’s Pregel [19] tackles a lot of the same graph analysis issues but uses its own more-expressive job distribution model. These systems focus on throughput for large tasks, rather than a high volume of updates and simple queries. Facebook has similar large-scale offline graph-processing systems that operate on data copied from TAO’s databases, but these analysis jobs do not execute within TAO itself.

## 10 Conclusion

Overall, this paper makes three contributions. First, we characterize a challenging Facebook workload: queries that require high throughput, low latency read access to the large, changing social graph. Second, we describe the objects and associations data model for Facebook’s social graph, and the API that serves it. Lastly, we detail TAO, our geographically distributed system that implements this API.

TAO is deployed at scale inside Facebook. Its separation of cache and persistent store has allowed those layers to be independently designed, scaled, and operated, and maximizes the reuse of components across our organization. This separation also allows us to choose different tradeoffs for efficiency and consistency at the two layers, and to use an idempotent cache invalidation strategy. TAO’s restricted data and consistency model has proven to be usable for our application developers while allowing an efficient and highly available implementation.

## Acknowledgements

We would like to thank Rajesh Nishtala, Tony Savor, and Barnaby Thieme for reading earlier versions of this paper and contributing many improvements. We thank the many Facebook engineers who built, used, and scaled the original implementation of the objects and associations API for providing us with the design insights and workload that led to TAO. Thanks also to our reviewers and our shepherd Phillipa Gill for their detailed comments.

## References

- [1] Amazon SimpleDB. <http://aws.amazon.com/simpledb/>.
- [2] Facebook – Company Info. <http://newsroom.fb.com>.
- [3] LevelDB. <https://code.google.com/p/leveldb>.
- [4] Project Voldemort. <http://project-voldemort.com/>.
- [5] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data system Research*, CIDR, 2011.
- [6] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating System Design and Implementation*, OSDI. USENIX Assoc., 2006.
- [7] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!’s hosted data serving platform. *PVLDB*, 1(2), 2008.
- [8] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google’s globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI, Berkeley, CA, USA, 2012.
- [9] F. Dabek, J. Li, E. Sit, J. Robertson, M. F. Kaashoek, and R. Morris. Designing a DHT for low latency and high throughput. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation*, NSDI, Berkeley, CA, USA, 2004.
- [10] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Voshall, and W. Vogels. Dynamo: Amazon’s highly available key-value store. In *Proceedings of 21st ACM Symposium on Operating Systems Principles*, SOSP, New York, NY, USA, 2007.
- [11] FlockDB. <http://engineering.twitter.com/2010/05/introducing-flockdb.html>.
- [12] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. Scalable consistency in scatter. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP, New York, NY, USA, 2011.
- [13] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, distributed data structures for internet service construction. In *Proceedings of the 4th Symposium on Operating System Design and Implementation*, OSDI, Berkeley, CA, USA, 2000.
- [14] U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: mining peta-scale graphs. *Knowledge Information Systems*, 27(2), 2011.
- [15] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent Hashing and Random trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proceedings of the 29th annual ACM Symposium on Theory of Computing*, STOC, 1997.
- [16] J. Li, J. Stribling, T. M. Gil, R. Morris, and M. F. Kaashoek. Comparing the performance of distributed hash tables under churn. In *Proceedings of the Third International Conference on Peer-to-Peer Systems*, IPTPS, Berlin, Heidelberg, 2004.
- [17] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don’t settle for eventual: scalable causal consistency for wide-area storage with COPS. In T. Wobber and P. Druschel, editors, *Proceedings of the 23rd ACM Symposium on Operating System Design and Implementation*, SOSP. ACM, 2011.
- [18] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, NSDI, 2013.
- [19] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In A. K. Elmagarmid and D. Agrawal, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, 2010.
- [20] Neo4j. <http://neo4j.org/>.
- [21] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, NSDI, 2013.
- [22] E. Nygren, R. K. Sitaraman, and J. Sun. The Akamai network: a platform for high-performance internet applications. *SIGOPS Operating Systems Review*, 44(3), Aug. 2010.
- [23] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In J. T.-L. Wang, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, 2008.
- [24] V. Ramasubramanian and E. G. Sirer. Beehive: O(1)lookup performance for power-law query distributions in peer-to-peer overlays. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation*, NSDI, Berkeley, CA, USA, 2004.
- [25] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM, New York, NY, USA, 2001.
- [26] Redis. <http://redis.io/>.
- [27] E. G. S. Robert Escriva, Bernard Wong. Hyperdex: A distributed, searchable key-value store for cloud computing. Technical report, Department of Computer Science, Cornell University, Ithaca, New York, December 2011.
- [28] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, Middleware, London, UK, UK, 2001.
- [29] M. Satyanarayanan. The evolution of coda. *ACM Transactions on Computer Systems*, 20(2), May 2002.
- [30] B. Shao and H. Wang. Trinity. <http://research.microsoft.com/en-us/projects/trinity/>.
- [31] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP, New York, NY, USA, 2011.
- [32] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM, New York, NY, USA, 2001.
- [33] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, SOSP, New York, NY, USA, 1995.
- [34] The Apache Software Foundation. <http://hbase.apache.org>, 2010.
- [35] W. Vogels. Eventually consistent. *Queue*, 6(6), Oct. 2008.