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

Nathan Bronson et al.

2013 Usenix Annual Technical Conference

# About Me

**Rohit Raveendran**

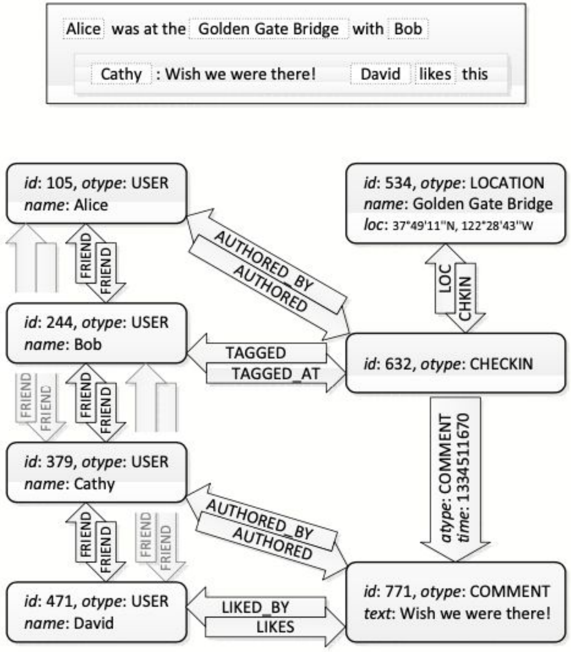
Principal Architect, Capillary Technologies

https://www.linkedin.com/in/rohit-raveendran-1529b1131/
https://www.twitter.com/rr0hit

# The Social Graph

- A flexible representation that **directly models real-life objects** and relationships
- **Objects** (Typed Nodes) connected by **Associations** (Typed Edges) form the Graph
- **Objects** model People, Places, Posts etc and repeatable Actions
- **Associations** model relationships, non-repeatable Actions and State Transitions
- **Associations** are directed, and often are tightly coupled with an inverse Edge

# The Social Graph



# The Social Graph

- Both Objects and Associations encode **data as Key Value pairs**; Associations have a time field
- The possible keys are defined by per-type schema
- The graph is **read-heavy**; Each page view involves fetching hundreds of items
- **Recent items** are often read the most
- **Read after Write consistency** is critical; **Eventual consistency** is acceptable

# The Pre-TAO Architecture

- The Social Graph is **stored in MySQL**, with masters and slaves distributed across “Regions”
- Memcached pools per “Region” function as a **lookaside cache** for Objects and Associations
- A **PHP abstraction** provides an Object-Association API without direct MySQL access for applications
- Data mapping, Cache invalidations and Control logic are implemented in the PHP client

# The Pre-TAO Architecture: Challenges

- Lookaside caches require a distributed control logic to avoid “Thundering Herds” *(Leases, Nishtala et al.)*
- Edge Lists cached in a Key-Value store have to be invalidated completely on updates.
- Harder Read After Write consistency *(Remote markers, Nishtala et al.)*
- Accessing the Graph data from non-PHP services

# TAO: The Objectives

- A **geographically distributed** graph storage system
- Serve **read queries from a "Region" local** to the requesting web server efficiently
- **Read and Write through** data access pattern, avoiding any storage layer access from clients
- Support **logic specific to the Social Graph** Eg. Adding inverse edges
- Favour **efficiency and availability over strong consistency**
- Ensure **Read-After-Write consistency** in the region where the write originated

# TAO: The API Characteristics

- Does not support a complete set of graph queries, but sufficient to serve the Social Graph use cases efficiently

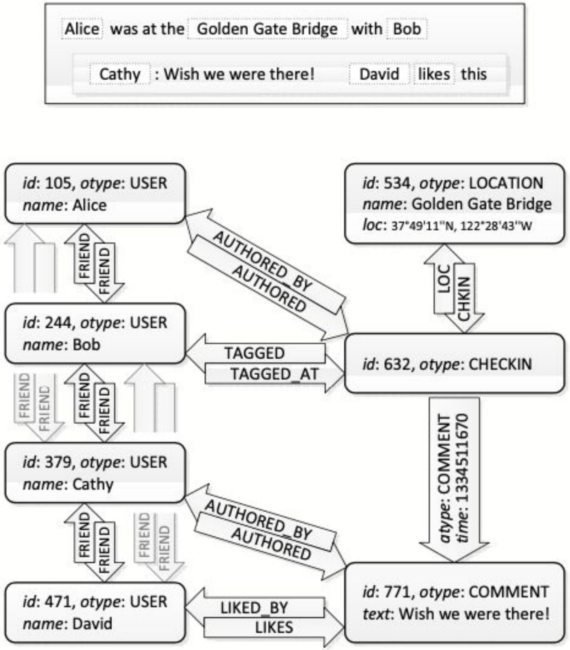
- The creation-time locality of Social Graph associations allows an association list to be always ordered by the time field. For example: Recent posts and comments are more often fetched than older ones

- Enforces a maximum number of associations per-association type to be returned for a query

# TAO: The API

- CRUD Operations on Objects

- A Restricted set of Association Operations
  - assoc_add(id1, atype, id2, time, (k→v)*)
  - assoc_delete(id1, atype, id2)
  - assoc_change_type(id1, atype, id2, newtype)
  - assoc_get(id1, atype, id2set, high?, low?)
  - assoc_count(id1, atype)
  - assoc_range(id1, atype, pos, limit)
  - assoc_time_range(id1, atype, high, low, limit)



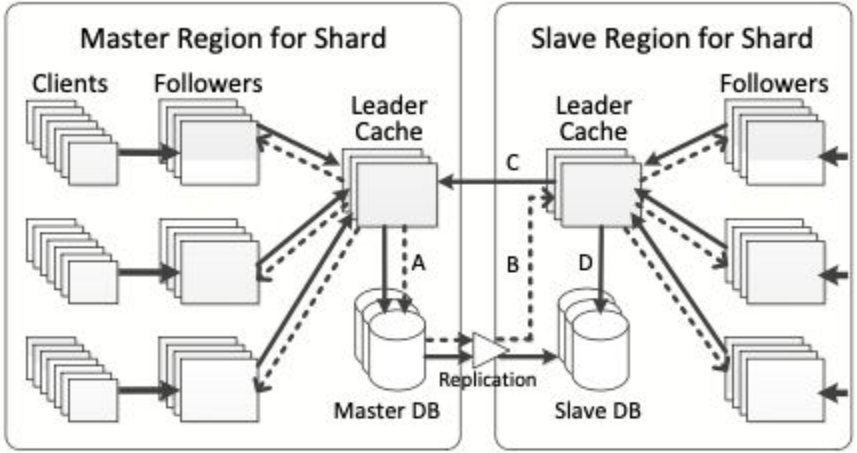
# TAO: The Storage Layer

- Objects and Associations are persisted in logically sharded MySQL databases
- Associations reside in the same shard as the `Object(id1)`
- Each MySQL Server is responsible for one or more shards
- Each shard manifests as a logical database within a MySQL server
- Physical servers are henced decoupled from the sharding technique

# TAO: The Storage Layer

- Shard to server mappings are modifiable to balance load across servers
- *Object Ids* contain an embedded *shard id* and are bound for its lifetime avoiding the need for a lookup
- Object attributes are serialized into a single `data` column. Access patterns involving filtering/searches do not get served by the Social Graph subsystem
- Database servers in a Region together have a full copy of the complete data, either in a master or slave role

# TAO: The Caching Layer



# TAO: The Caching Layer

- The caching layer implements the TAO API
- Multiple caching servers form a *Cache Tier* that is capable of serving any TAO request
- Storage layer shards are mapped to cache servers using consistent hashing
- A single Leader Tier is responsible for all storage layer reads and writes

# TAO: The Caching Layer

- A region can have several *Follower Tiers* that serve the client requests
- On cache misses and write requests, followers contact the Leader tier
- Leader tier issues asynchronous invalidation/refill messages to its followers
- Cache servers evict items using a least recently used (LRU) policy
- Writes create inverse associations as it is type-aware, but without any atomicity guarantees

# TAO: The Leader Tier

- Leaders perform all the reads from the local Storage Layer
- Leaders in master regions handle the writes to the storage layer, issues invalidation/refill messages to its followers
- Leaders in master regions handling a cross-region write requests, additionally return change sets to the requesting leader to propagate to the requesting follower
- Leaders in master regions embed invalidation/refill messages as writes to a black-hole table in the storage layer (referred to as embedded messages)
- Leaders in slave regions for a storage layer shard, delegates the write requests to the Leader tier of the master region

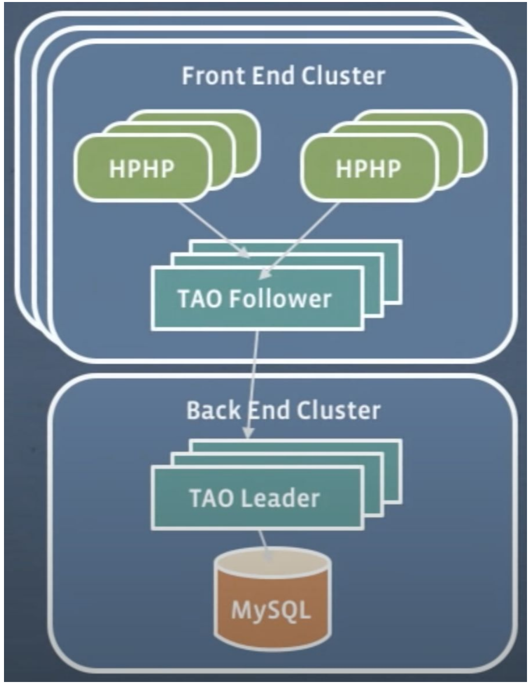
# TAO: The Leader Tier

- Leaders in slave regions subscribe to embedded invalidation/refill messages from the replication logs and in turn deliver them to the local followers
- In case of failure to deliver invalidation/refill messages to the followers, leader queues them into disk
- Leaders acting as the central cache coordinator prevents Thundering Herds by serializing concurrent writes for the same key
- Leaders optimize the cross-region communications by bundling multiple requests into a single RPC call

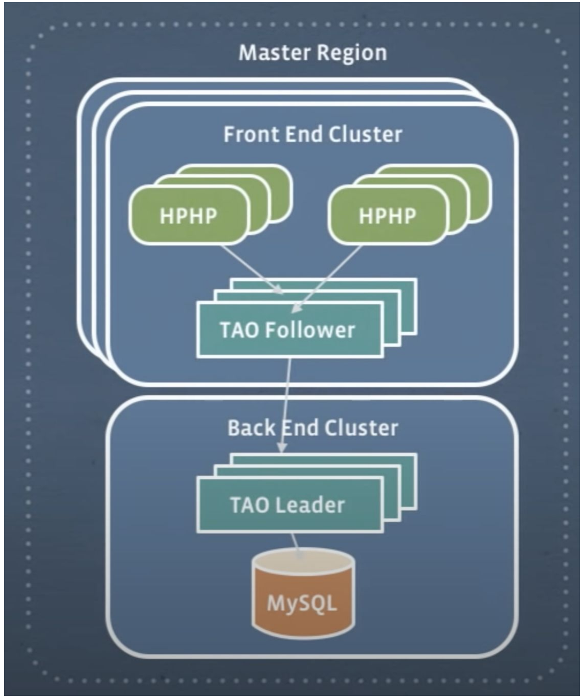
# TAO: Follower Tier

- Followers serve the clients via the TAO API
- Followers delegate cache misses and writes to the leader
- In case of writes, the follower tier applies the changeset returned by the leader synchronously to achieve Read-After-Write consistency in the same tier
- Followers process invalidation/refill messages from the Leader
- Followers perform shard cloning to avoid hot spots i.e overloading specific cache servers housing a "popular" key

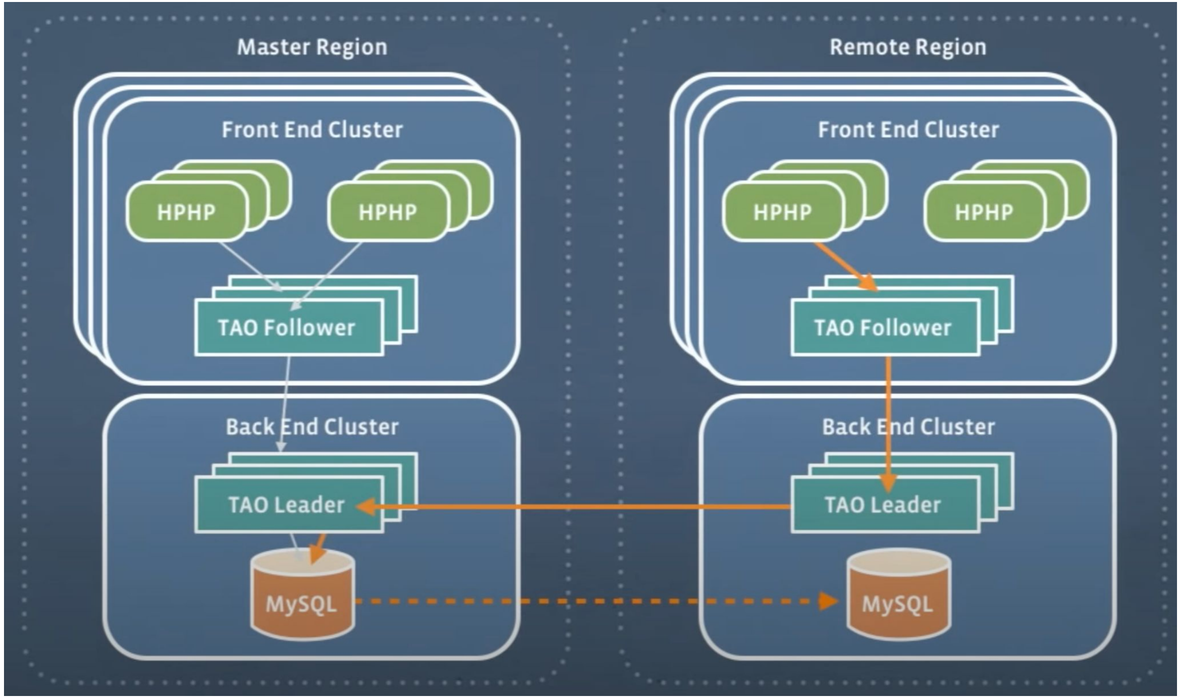
# Tao: The Read Path



# TAO: The Write Path (Master Region)



# TAO: Write Path (The Slave Region)



# TAO: Consistency Model

- TAO is eventually consistent owing to the asynchronous replication behaviour across regions and across followers
- Read-After-Write consistency is assured in the same tier owing to the synchronous delivery of the changeset applied. Changesets are only applied after verifying that current “version” of the data in the cache matches the pre update value in the changeset.
- Inconsistencies may arise in case of partial leader failure or follower failures during writes which resolve themselves on receiving the invalidation/refill messages
- In case of a cache eviction due to indications in changeset data version, clients may observe a  go back in time if the storage layer replication takes longer than the cache eviction

# TAO: Failure Handling

- A **storage layer master failure** is handled by auto promoting an existing slave
- A **storage layer slave failure** is handled by redirecting follower reads to the leader in the shard's master region. Invalidation/refill messages are delivered from the master region
- On **slave promotion**, the embedded invalidation/refill messages may be replayed in all regions to ensure data consistency

# TAO: Failure Handling

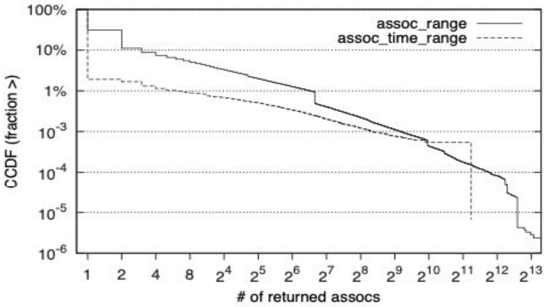
- On a **leader cache server failure**, followers handle cache misses by reading directly off the storage layer
- On a **leader cache server failure**, writes are sent to a replacement leader chosen at random, which queues up embedded invalidation/refill messages for the original leader until it is back in service
- **Refill and invalidation failures** are handled by queuing the messages to the disk to be delivered later
- **Follower server failures** are handled by the client by routing requests to a backup follower tier

# TAO: Workload and Performance

- Only 0.2% of the requests are writes

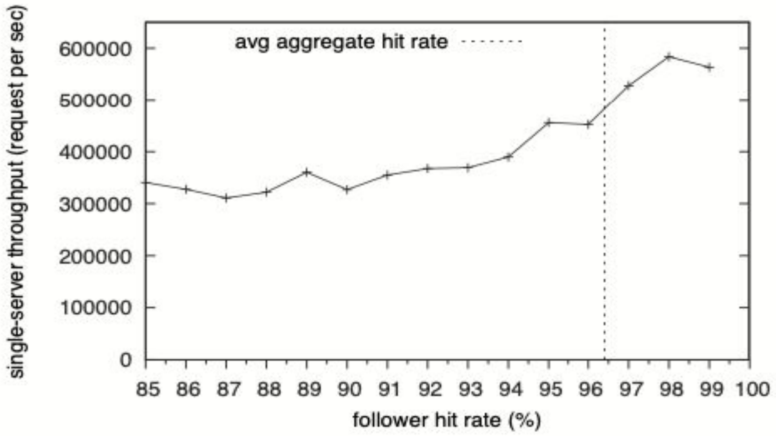
| read requests | 99.8 % | write requests | 0.2 % |
|---|---|---|---|
| assoc_get | 15.7 % | assoc_add | 52.5 % |
| assoc_range | 40.9 % | assoc_del | 8.3 % |
| assoc_time_range | 2.8 % | assoc_change_type | 0.9 % |
| assoc_count | 11.7 % | obj_add | 16.5 % |
| obj_get | 28.9 % | obj_update | 20.7 %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

- Association lists are very often empty or small



# TAO: Workload and Performance

- Throughput: Half a million rps single-server throughput at maximum hit rate




- Availability: Failed queries account for 0.00049% of total received requests

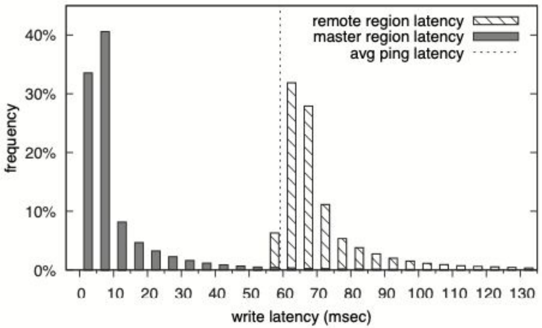
# TAO: Workload and Performance

- Overall hit rate of 96.4%
- Read Latencies

| | *hit lat. (msec)* | | | *miss lat. (msec)* | | |
|---|---|---|---|---|---|---|
| *operation* | *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 |

# TAO: Workload and Performance

- Write Latency




- Slave storage servers lag their master by less than 1 second during 85% of the sampling window, by less than 3 seconds 99% of the time, and by less than 10 seconds 99.8% of the time

# Related Work

- Eventual consistency & read-after-write consistency variants as presented by Terry et al (SOSP, 1995) and Vogels (ACM Queue, 08)
- Geographically distributed data stores like Google's Spanner that distributes data across geographies ensuring one leader shard using the Paxos algorithm
- Distributed hash tables and key-value systems like Amazon DynamoDB
- Hierarchical connectivity like Akamai content cache systems that minimises latencies by avoiding cross-region requests
- Structured storage systems like SimpleDB with weaker guarantees than traditional RDBMS
- Modern Graph databases like Neo4j

# Summary

- The Social Graph data model and its characteristics are presented
- TAO introduces hierarchical data layers to optimize for latencies
- TAO introduces an asynchronous model to optimally ensure data consistency across geographies

# References

- TAO: Facebook's Distributed Data Store for the Social Graph
- Scaling Memcache at Facebook
- USENIX ATC '13 - TAO: Facebook’s Distributed Data Store for the Social Graph
- Large-Scale Low-Latency Storage for the Social Network - Data@Scale
- Eventually Consistent - Revisited

# Leases

- Addresses the issues of “**Thundering Herds**” and “**Stale Sets**”
- **Thundering Herds** occur when a specific key undergoes heavy read and write activity. Repeated invalidations cause several reads to default to costly reads from the storage layer
- **Stale Sets** occur when a client sets an older value in to cache due to concurrent writes
- Cache server issues a lease (a token bound to the key) when client initiates a set request
- Cache server invalidates a lease when a key is deleted (Update is a delete and set for simple caches)
- A set request is validated against the lease issued to prevent **Stale Sets**
- For a short duration after issual of a lease for a key, reads to the key are sent a special message instructing retrial after a small time window (few milliseconds). This mitigates **Thundering Herds**

# Remote Markers

- Storage layer replication of data needs to be ensured before invalidating caches in a slave region to prevent stale sets
- When a client writes to a key K, it sets a marker Rk in the local storage layer
- The write in the master region embeds the deletion of Rk in the write query
- On reads in the slave region, requests are routed to the master region as long as an Rk is present