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

Before TAO

Data stored in MySQL and cached in memcached

Caching managed by apps

Problems:

- Operations on lists are inefficient in memcached (update whole list)
- Complexity of clients managing cache
- Hard to offer read-after-write consistency

What is TAO?

TAO is a storage system for graphs that manages both durability and caching

- Unlike memcached, it's a **write-through** cache
- Unlike both memcached and MySQL, API explicitly involves graph concepts

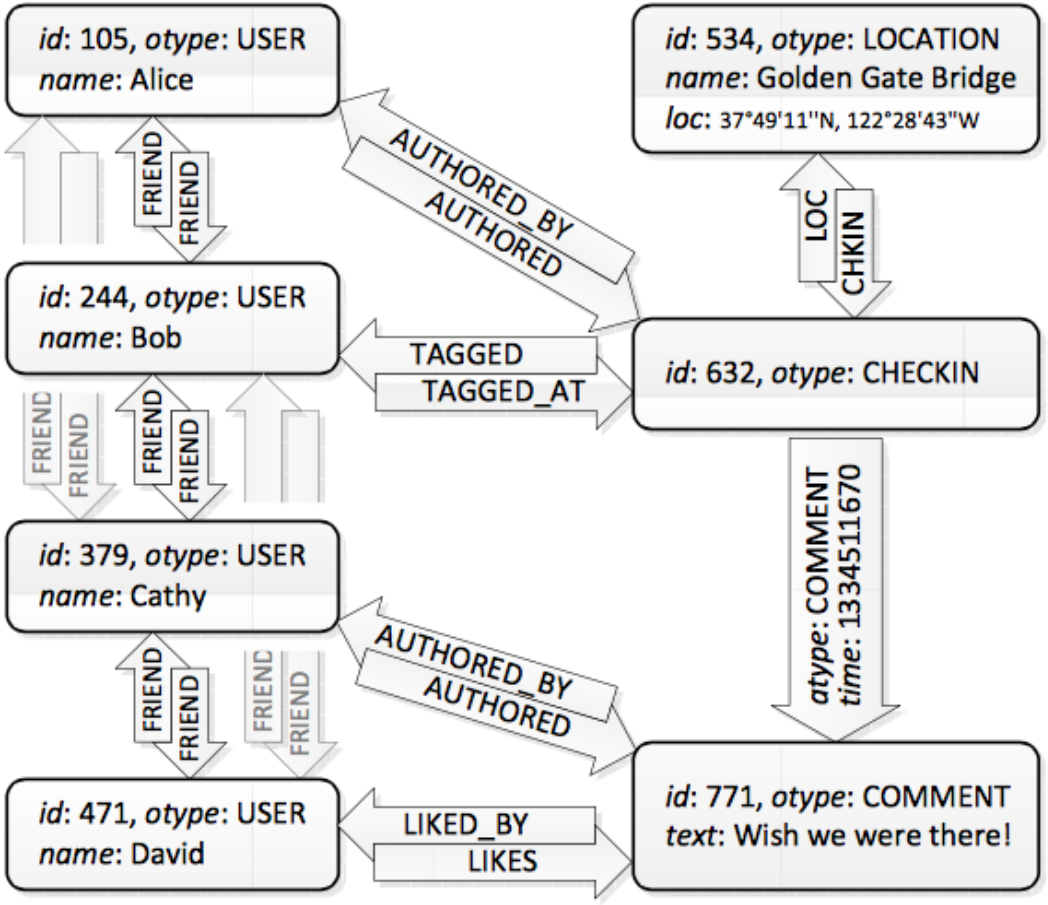
Eventually consistent, but "read-after-write" in many common cases

Data Model

Objects (e.g. user, place) with unique IDs

Associations (e.g. tagged) between two IDs

Both have key-value data as well as a time field



API

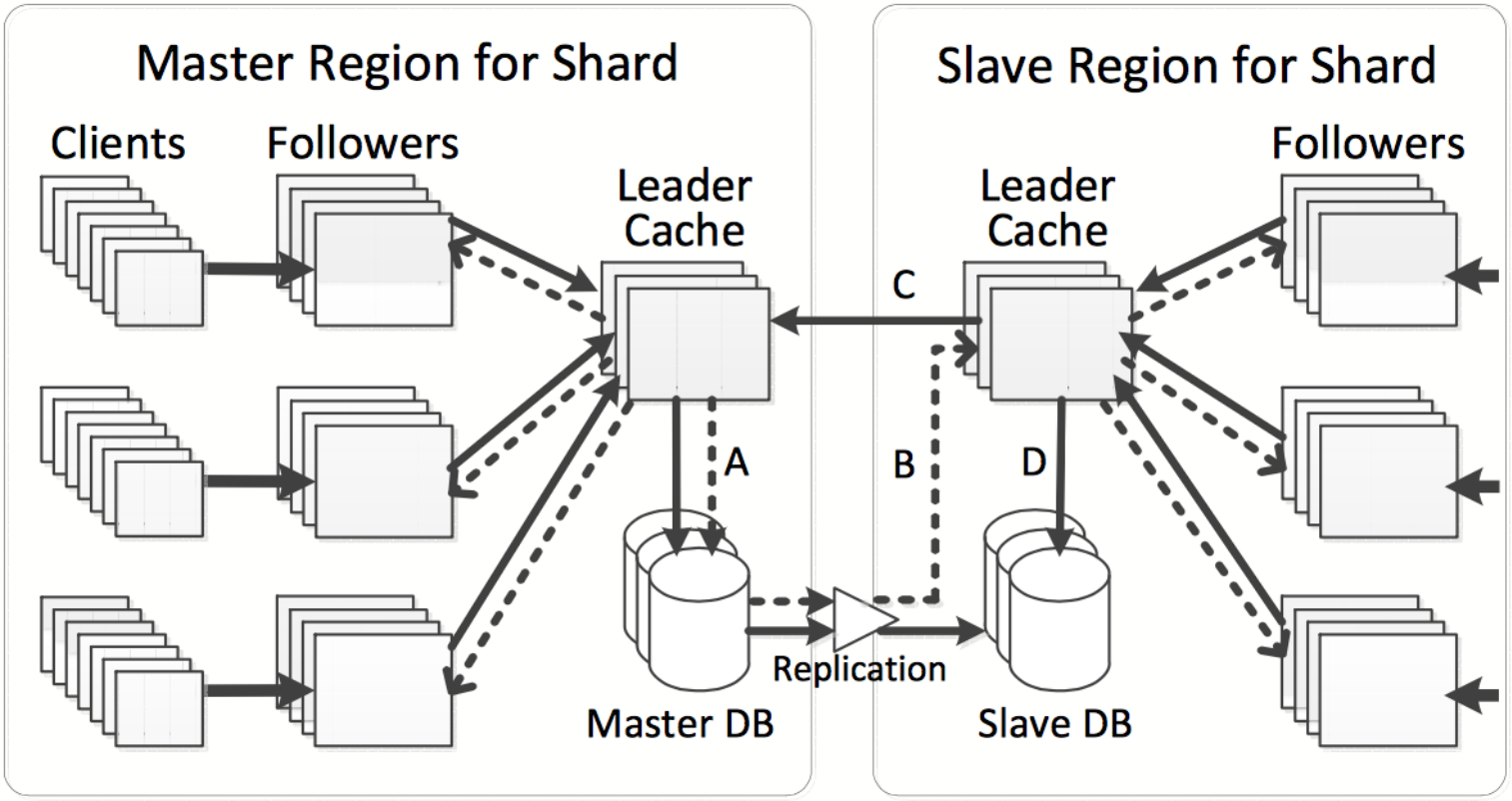
Create/delete/update objects and assocs

Association queries:

- assoc_get(id1, atype, id2set, high?, low?) ➜ list assocs between specific IDs
- assoc_count(id1, atype) ➜ count assocs
- assoc_range(id1, atype, pos, limit) ➜ list all assocs by position
- assoc_time_range(id1, atype, high, low, limit) ➜ list assocs in time range

After use, all results are filtered via privacy controls

Architecture



MySQL databases → durability

Leader cache → coordinates writes to each object

Follower caches → serve reads but not writes

Partitioning

Objects are allocated to fixed “shards” via their object ID; these may move across databases, etc after creation

Associations (id1, atype, id2) stored on same shard as id1

One leader cache server is responsible for each shard and its assocs

Leader and Follower Caches

Why two cache tiers?

- Quadratic growth in all-to-all connections for a single tier (because each server has to send writes to the server for their object ID)
- Hot spots
- Read-dominated workload

Behavior

- All writes go to local and then master-region leaders
- These send updates to followers and the local leader that started the write (so that its clients can immediately see it)
- Updates across regions piggy-back on MySQL replication log
- Some possibility for inconsistency here

Consistency

Consistency is eventual only; some reasons include:

- Master-slave replication of MySQL
- Cache followers can be behind leaders
- Some writes touch two items (e.g. maintaining inverse edges)

However, TAO does try to provide “read-after-write” consistency when at most one failure occurs, by updating caches on the request path in-place

- Local leader, local follower, master leader, master MySQL

Workload

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 %

Very read-heavy

Most edge queries have empty results

Long tails in most distributions