

L. Lancia, G. Salillari

Cloud Computing

Master Degree in Data Science

Sapienza Università di Roma

# Facebook Tao

Distributed Data Store for the Social Graph

# Table of Contents

The Data Model

Architecture

Implementation

Consistency & Failures

Workload & Performance

# Introduction

## What is TAO?

TAO is a geographically distribute store

- deployed at Facebook
- with efficient and timely access to social graph
- using a fixed set of query
- replacing memcache
- running on thousands of machines
- provide access to many PB of data
- process a billion reads ad millions of writes each second!

## The social graph

Facebook has more than 1 billion active user

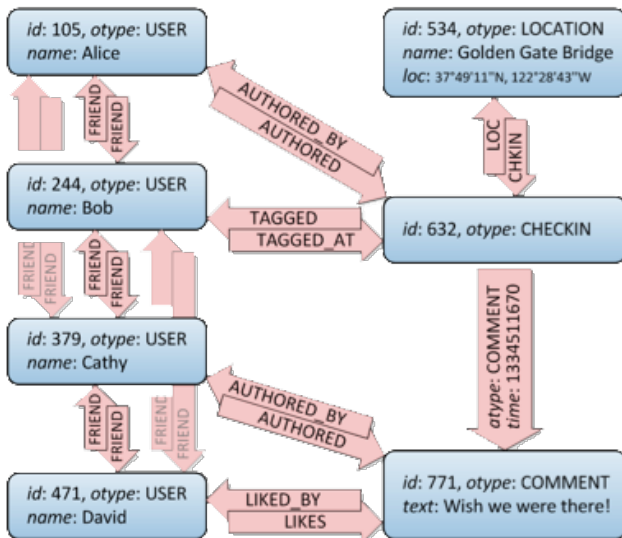
- recording relationships,
- sharing interests,
- uploading pictures and ...

The user experience of Fb comes from rapid, efficient and scalable access to the *social graph*

What's behind an entry in yours Fb page?



A single Fb page aggregate and filter hundreds of items from the social graph.



## Before Tao

- Facebook was storing the social graph to MySQL
  - Querying it from PHP
  - Storing result in memcache



Over time Fb deprecated direct access to MySQL in favor of a graph (associations, nodes) abstraction

## Limits

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

Also they want to access social graph from non-PHP services



# TAO's Goals

- **Efficiency at Scale**
- Low read latency
- Timeliness of writes
- High read availability

## TAO's Goals

- Efficiency at Scale
- Low read latency
- Timeliness of writes
- High read availability

## TAO's Goals

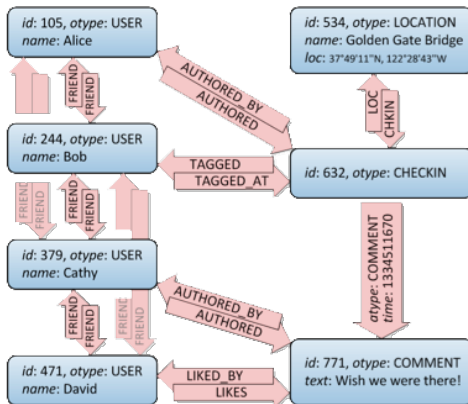
- Efficiency at Scale
- Low read latency
- Timeliness of writes
- High read availability

## TAO's Goals

- Efficiency at Scale
- Low read latency
- Timeliness of writes
- High read availability

# Tao Data Model

T.A.O. stands for “The Associations and Objects”



# Objects

- Typed nodes (type is denoted by `otype`)
- Identified by 64-bit integers (unique)
- Contains data in the form of key-value pairs
- Models users and repeatable actions (e.g. comments)

API for objects:

- Allocate new object
- retrieve
- update
- delete

# Objects

- Typed nodes (type is denoted by `otype`)
- Identified by 64-bit integers (unique)
- Contains data in the form of key-value pairs
- Models users and repeatable actions (e.g. comments)

API for objects:

- Allocate new object
- retrieve
- update
- delete

## Associations

- Typed directed edges between objects (type is denoted by `atype`)
- Identified by source object `id1`, `atype` and destination object `id2`
- Contains data in the form of key-value pairs.
- Contains a 32-bit `time` field.
- Models actions that happen at most once or records state transition (e.g. like)
- Often inverse association is also meaningful (eg like and liked by).



# Associations API

- Add new
- Delete
- Change type

Also inverse association is created or modified automatically

## Querying TAO

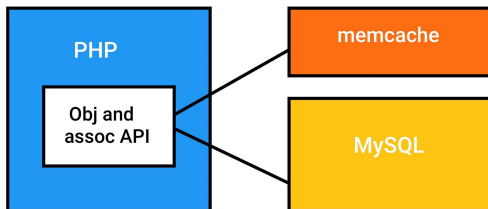
TAO's associations queries are organized around *associations list*

- `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)`

Query results are bounded to 6000 results

# Architecture

## Before Tao

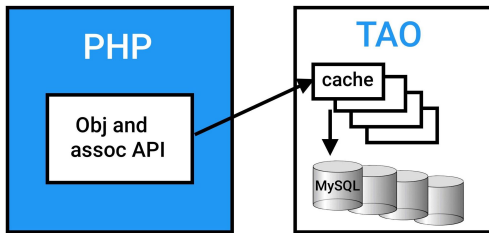


## After Tao

# Architecture

Before Tao

After Tao



## Storage Layer

- Object and Associations are stored in MySQL (before & with TAO)
- TAO API is mapped to a small set of SQL queries
- A single MySQL server can't handle TAO volumes of data
  - We divide data into logical *shards*
  - *shards* are mapped to db
  - different servers are responsible for multiple shards
  - mapping is adjusted for load balancing
- Object are bounded to a *shard* for their entire lifetime
- Associations are stored in the *shard* of its `id1`

## Cache Layer

### TAO cache

- contains: Objects, Associations, Associations counts
- implement the complete API for clients
- handles all the communication with storage layer
- it's filled on demand and evict the least recently used items
- Understand the semantic of their contents

It consists of multiple servers forming a *tier*

- Request are forwarded to correct server by a *sharding* scheme as dbs
- For cache miss and write request, the server contacts other caches or db

## Yet Another caching layer

**Problem:** A single caching layer divided into a *tier* is susceptible to *hot spot*

**Solution:** Split the caching layer in two levels

- A *Leader* tier
- Multiple *Followers* tiers

## Yet Another caching layer

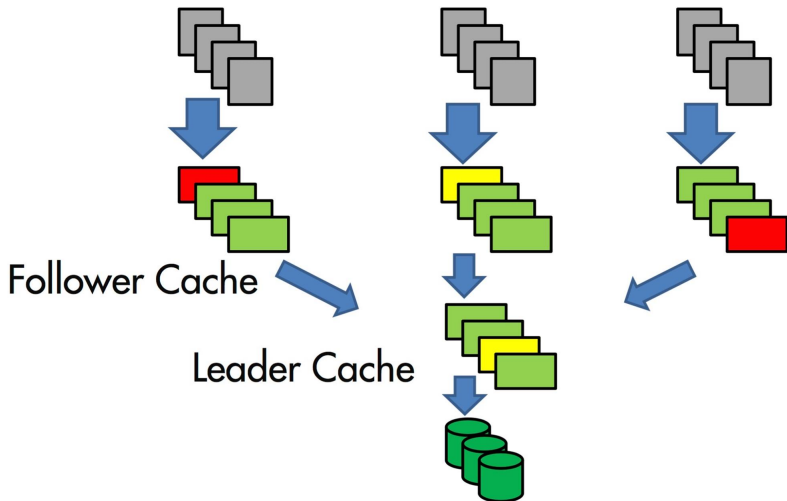
**Problem:** A single caching layer divided into a *tier* is susceptible to *hot spot*

**Solution:** Split the caching layer in two levels

- A *Leader* tier
- Multiple *Followers* tiers



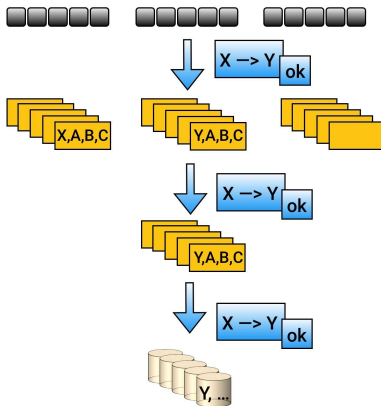
## Leaders & Followers



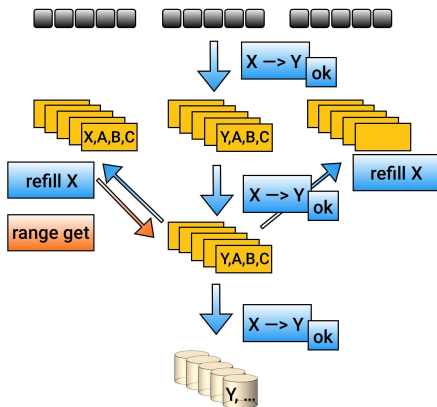
## Leaders & Followers

- Followers forward all writes and read cache misses to the leader tier
- Leader sends async cache maintenance messages to follower tier
  - Eventually Consistent
- If a follower issues a write, the follower's cache is updated synchronously
- Each update message has a version number
- Leader serializes writes

# Leaders & Followers



# Leaders & Followers



## Scaling Geographically

**Problem:** Network latencies are not low in a multi Data Centers environment

Considering that read misses are more common than writes in the follower tier

**Solution:** Handles read cache miss locally

## Scaling Geographically

**Problem:** Network latencies are not low in a multi Data Centers environment

Considering that read misses are more common than writes in the follower tier

**Solution:** Handles read cache miss locally

## Scaling Geographically

**Problem:** Network latencies are not low in a multi Data Centers environment

Considering that read misses are more common than writes in the follower tier

**Solution:** Handles read cache miss locally

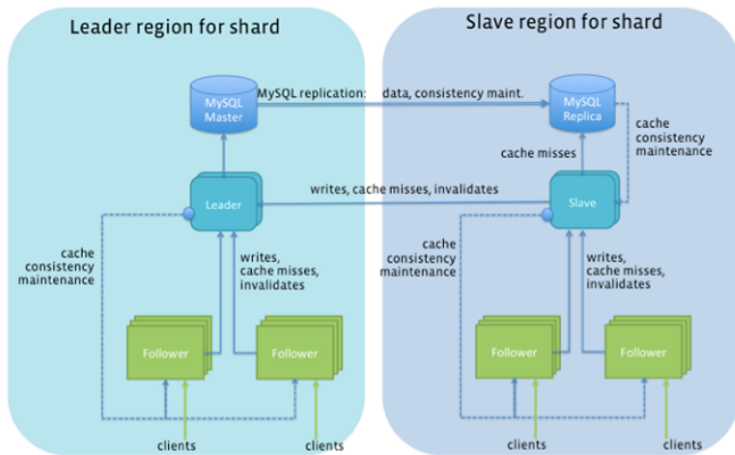
## Master & Slave Regions

Fb cluster together DC in regions (with low intra region latency)

- Each region have a full copy of the social graph
- Region are defined master or slave for each shard
- Followers send read misses and write requests to the local leader
- Local leaders service read misses locally
- Slave leaders forward writes to the master shard
- The slave leader will update it's cache ahead of the async updates to the persistent store



# Overall Architecture



# Implementation

To achieve performance and storage efficiency Fb have implemented some optimizations to servers and DBMS's.

## Caching Servers

- Memory is partitioned into arenas by association type
  - This mitigates the issues of poorly behaved association types
  - They can also change the lifetime for important associations
- Small items with fixed size have a lot of pointer overhead
  - They use a directly mapped 8-way LRU cache
  - Used for association counts

## MySQL Mapping

We divided the space of objects and associations into *shards*.  
Each *shard*:

- is assigned to a logical DB
- there is a table for objects and a table for associations
- all field of object are serialized in a single data column
- object of different size can be stored in the same column

Exceptions:

- Some object can benefit from being stored in a different table
- Associations counts are stored in a separate table

## Cache Sharding

Shards are mapped to cache server using consistent hashing (like dynamo)

This can lead to *imbalances*, so TAO use shard cloning to rebalance the load

There are also *popular object* that can be queried a lot more often than others.

TAO says to the clients to cache them these objects

## High-Degree Objects

Some object have a lot of associations (remember there were a limit of 6000?)

- TAO can't cache all associations list
- Requests will always end to Db

so

- For `assoc_count`, the edge direction is chosen using the lower degree between source and destination object
- For `assoc_get` query, only associations whose `time > object's creation time`

## Consistency

Under normal operation, TAO is *eventually consistent*

Replication lag usually  $< 1''$

Race conditions are resolved by using version numbers

In special "*critical*" situation a read can be forwarded to database to ensure to read from a consistent source of truth.  
(Useful for auth procedures)

Read after write consistency cannot be ensured during fails for a short time

## Detecting Failures

Each TAO server stores per-destination time-outs

- if several time-outs occurs, hosts are marked as down
- subsequent requests are aborted
- Tao reacts trying to route around failures (favouring availability over consistency)
- Down hosts are actively probed to check if recover



## Handling Failures

**Database Fail** Db can crash or be off-line for maintenance.

- If master db is down, a slave is promoted to new master
- If a slave db is down, cache miss are redirected to TAO leaders in master region

**Leader Fail** Followers re-route requests around it

- Read miss goes directly to db
- Write miss are routed to a random member of the leader tier

## Handling Failures (2)

**Invalidation Fail** Leader can't contact a follower during a cache invalidation message

- Leader queues message
- If Leader also crash message are lost so new leader send bulk invalidation

**Follower Fail** Followers in others tiers share the responsibility of it's shard

- Tao client have a primary tier and a backup tier

## Workload

<b>read requests</b>	<b>99.8 %</b>	<b>write requests</b>	<b>0.2 %</b>
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 %

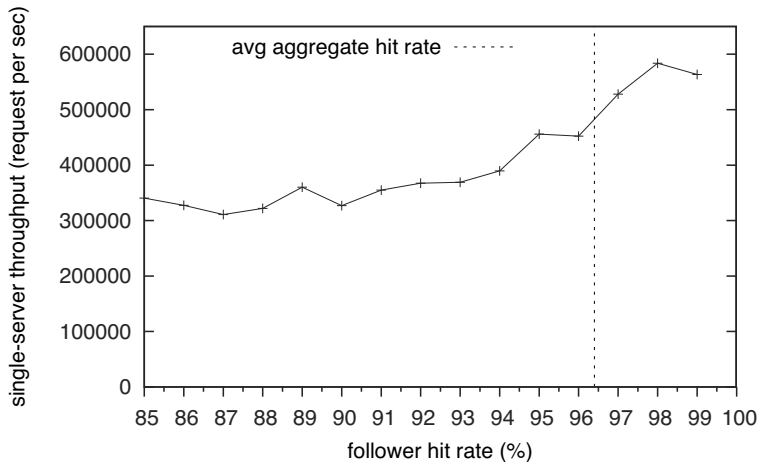
Frequencies for client request

## Availability

Under real workload, over a period of 90 days, the **fraction of failed TAO queries** is:

$$4.9 \times 10^{-6}$$

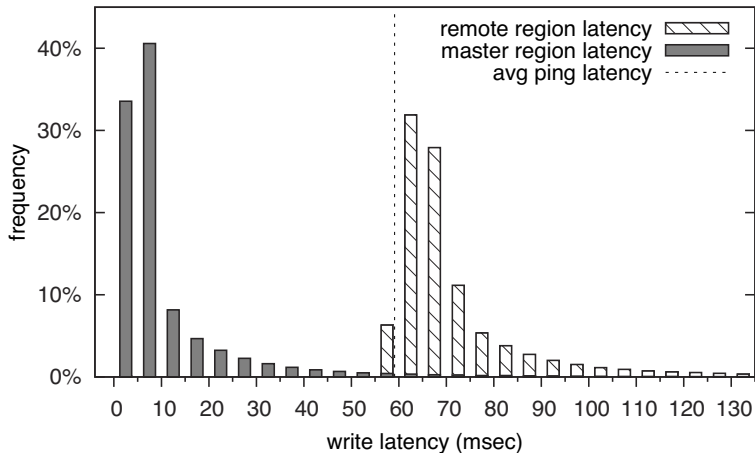
# Followers Capacity



## Hit Rates and latency

<i>operation</i>	<i>hit lat. (msec)</i>			<i>miss lat. (msec)</i>		
	<i>50%</i>	<i>avg</i>	<i>99%</i>	<i>50%</i>	<i>avg</i>	<i>99%</i>
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

# Write Latency



## Summarizing

### Read latency

- Separate cache from database
- Graph aware cache

### Efficiency at scale

- Subdividing Data Centers

### Write timeliness

- Write trough cache
- Async replication

### Read availability

- Multiple data sources



# Thank You