TAO
Facebook’s Online Graph Store

Nathan Bronson

CS244b–11 Nov 2014
The Social Graph

At the summit — at Charlotte Dome.

PHOTO

LOCATION

GPS_DATA

AT

CHECKIN

POST

AUTHOR

FRIEND

USER

USER

CAROL

USER

ALICE

Like · Comment

Carol and 3 others like this.

Alice

how was it? is it worth the long approach?

Like · More · Just now

Write a comment...

(hypothetical encoding)
Dynamically Rendering the Graph

Web Server (PHP)
Dynamically Rendering the Graph

TAO

Web Server (PHP)

• >1 billion queries/second
• many petabytes of data
What Are TAO's Goals/Challenges?

- Efficiency at scale
Data Dependencies and Query Rounds

- Thousands of queries to render news feed
- Dependence chains several dozen queries long
- ~500 reads per write
What Are TAO’s Goals/Challenges?

- Efficiency at scale
- Low read latency
- Timeliness of writes
- High Read Availability
Graph in Memcache

Web Server (PHP)

Obj & Assoc API

memcache
(nodes, edges, edge lists)

mysql

Web Server (PHP) communicates with Memcache and MySQL through an object and associative API.
Objects = Nodes

- Identified by unique 64-bit IDs
- Typed, with a schema for fields

id: 1807 =>
  type: POST
  str: “At the summ...”

id: 2003 =>
  type: COMMENT
  str: “how was it ...”

id: 308 =>
  type: USER
  name: “Alice”

Associations = Edges

- Identified by \(<id_1, \text{type}, id_2>\)
- Bidirectional associations are two edges, same or different type

\(<1807, \text{COMMENT}, 2003>\)
  time: 1,371,704,655

\(<308, \text{AUTHORED}, 2003>\)
  time: 1,371,707,355

\(<2003, \text{AUTHOR}, 308>\)
  time: 1,371,707,355
Association Lists

- `<id1, type, *>`
- Descending order by time
- Query sublist by position or time
- Query size of entire list

```
id: 1807 => type: POST
str: “At the summ...
```

```
<1807,COMMENT,4141>
time: 1,371,709,009
str: “Been wanting to do ..."
```

```
<1807,COMMENT,8332>
time: 1,371,708,678
str: “The rock is flawless, ...
```

```
<1807,COMMENT,2003>
time: 1,371,707,355
str: “how was it, was it w...
```

```
id: 4141 => type: COMMENT
str: “Been wanting to do ..."
```

```
id: 8332 => type: COMMENT
str: “The rock is flawless, ...
```

```
id: 2003 => type: COMMENT
str: “how was it, was it w...
```
Objects and Associations API

Reads – 99.8%

- Point queries
  - `obj_get`: 28.9%
  - `assoc_get`: 15.7%

- Range queries
  - `assoc_range`: 40.9%
  - `assoc_time_range`: 2.8%

- Count queries
  - `assoc_count`: 11.7%

Writes – 0.2%

- Create, update, delete for objects
  - `obj_add`: 16.5%
  - `obj_update`: 20.7%
  - `obj_del`: 2.0%

- Set and delete for associations
  - `assoc_add`: 52.5%
  - `assoc_del`: 8.3%
What Are TAO’s Goals/Challenges?

- Efficiency at scale
- Low read latency
- Timeliness of writes
- High Read Availability
Mapping FBIDs to machines

- Two-level mapping
  - Fixed n-to-1 mapping from fbid $\rightarrow$ shard
  - Dynamic n-to-1 mapping from shard $\rightarrow$ machine

- Advantages:
  - Efficient and scalable
  - Allows good load balancing
  - Allows explicit colocation or anti-colocation

- Disadvantages:
  - Colocation is forever
Independent Scaling by Separating Roles

Web servers

Cache
- Objects
- Assoc lists
- Assoc counts
- Presence + absence!

Database

• Stateless

• Sharded by id
• Servers -> read qps

• Sharded by id
• Servers -> bytes

TAO
Subdividing the Data Center

- Web servers
  - Inefficient failure detection
  - Many switch traversals
- Cache
  - Many open sockets
  - Lots of hot spots
- Database
Subdividing the Data Center

Web servers

Cache

- Distributed write control logic

Database

- Thundering herds
Follower and Leader Caches

Web servers

Follower cache

Leader cache

Database
Avoiding Long-Latency Reads

Full Database Replicas
What Are TAO's Goals/Challenges?

- Efficiency at scale
- Low read latency
- Timeliness of writes
- High Read Availability
Write-through Caching – Association Lists

Web servers

Follower cache

Leader cache

Database

X, A, B, C
X → Y
ok
refill X

Y, A, B, C

range get

Y, A, B, C

X → Y
ok
refill X

Y, ...

X → Y
ok
refill X

X → Y
ok
Asynchronous DB Replication + Local Read-What-You-Wrote

Web servers

Master data center

Replica data center

Inval and refill embedded in SQL

 Writes forwarded to master

Delivery after DB replication done

Cache gets new value immediately
What Are TAO’s Goals/Challenges?

- Efficiency at scale
- Low read latency
- Timeliness of writes
- High Read Availability
Improving Availability: Read Failover

Master data center

- Web servers
- Follower cache
- Leader cache
- Database

Replica data center

- Web servers
- Follower cache
- Leader cache
- Database
Move Fast

Efficiency at scale
- Separate cache and DB
- Graph-specific caching
- Efficient negative caches
- Subdivide data centers

Read latency

Write timeliness
- Write-through cache
- Asynchronous replication

Read availability
- Alternate data sources
Move Fast + Break Things – Consistency

- No causal consistency, even without races
- Partial write failures – writes are slow so they must be batched, which means cleanup code needs to handle $2^{\text{num\_shard}}-1$ failure cases
- No atomicity – client code must always be robust to partial writes

Simple mental model is almost as good as strong semantics

Can we make strong semantics pay-for-what-you-use?

If we choose causal+ consistency, how do we thread happens-before through the browser?
Move Fast + Break Things – MegaObjects

- 3M QPS for a single object – probably an object used for configuration, and maybe a SPOF
- 800M out-edges for a single object – but most edge lists are of size 1. Eventual size can’t be predicted at object creation time
- 50M out-edges with the same time – many of the original tools used time as a cursor

*Apps will expand the original use cases until the system breaks (and more)*

*Can we keep everything in one system? Very valuable*
Move Fast and Break Things – Simple API

- No compare-and-set – racing writers aren’t atomic
- 3-way relationships – how do I link my USER node with a song I’ve listened to multiple times?
- Document storage – many objects hold JSON, could we optimize updates to individual sub-documents?
- Indexes – managed by external systems, so they can become inconsistent

Simplicity and decoupling are essential for reliability, but push complexity elsewhere. What is the right tradeoff?
Questions?

Efficiency at scale
- Separate cache and DB
- Graph-specific caching
- Efficient negative caches
- Subdivide data centers

Read latency

Write timeliness
- Write-through cache
- Asynchronous replication

Read availability
- Alternate data sources
Inverse associations

- Bidirectional relationships have separate $a \rightarrow b$ and $b \rightarrow a$ edges
  - $\text{inv}_\text{type}(\text{LIKES}) = \text{LIKED\_BY}$
  - $\text{inv}_\text{type}(\text{FRIEND\_OF}) = \text{FRIEND\_OF}$
- Forward and inverse types linked only during write
  - TAO assoc_add will update both
- Not atomic, but failures are logged and repaired
Single-server Peak Observed Capacity

Hit rate

Operations/second

90% 91% 92% 93% 94% 95% 96% 97% 98% 99%

0 K 100 K 200 K 300 K 400 K 500 K 600 K 700 K
Write latency
More In the Paper

▪ The role of association time in optimizing cache hit rates
▪ Optimized graph-specific data structures
▪ Write failover
▪ Failure recovery
▪ Workload characterization