Graph stores in the wild
Graph stores in the wild

Zen: Pinterest’s Graph Storage Service
Graph stores in the wild

Zen: Pinterest’s Graph Storage Service

twitter / flockdb

A distributed, fault-tolerant graph database
Graph stores in the wild

Zen: Pinterest’s Graph Storage Service

twitter / flockdb

A distributed, fault-tolerant graph database

LinkedIn’s GraphDB
Graph stores in the wild

LinkedIn’s GraphDB

Zen: Pinterest’s Graph Storage Service

twitter / flockdb

A distributed, fault-tolerant graph database
Graph stores in the wild

Zen: Pinterest’s Graph Storage Service

Twitter / flockdb

A distributed, fault-tolerant graph database

LinkedIn’s GraphDB

neo4j

TITAN
Graph stores in the wild

Zen: Pinterest’s Graph Storage Service

twitter / flockdb

A distributed, fault-tolerant graph database

LinkedIn’s GraphDB
Graph stores in the wild

Zen: Pinterest's Graph Storage Service

neo4j

A distributed, fault-tolerant graph database

TITAN

LinkedIn's GraphDB

Key diff. from Graph Processing: user-facing!
Graph stores in the wild

Zen: LinkedIn’s Graph Storage Service

neo4j

twitter / flockdb

TITAN

Very active space, both in industry & academia

Key diff. from Graph Processing: user-facing!
Graph stores in the wild

- LinkedIn’s GraphDB
- Neo4j

Huge variance in scale and approach

Very active space, both in industry & academia

Key diff. from Graph Processing: user-facing!
Problem

User-facing serving of a billion-node, trillion-edge social graph
• FB full graph in $O(\text{petabyte})$, not gonna fit in my laptop

Extremely high read load, due to freshness & privacy filtering
• sustained $> \text{one billion queries per second}$

Previous approach: lookaside memcache + MySQL:
1. KV pair is inefficient
2. expensive read-after-write consistency
Data Model

Object: \((id) \rightarrow (otype, (key \rightarrow value)*)\)
Assoc.: \((id1, atype, id2) \rightarrow (time, (key \rightarrow value)*)\)
Association List: \((id1, atype) \rightarrow [a_{new} \ldots a_{old}]\)
API

assoc_range(src, atype, off, len)
obj_get(nodeId)
assoc_get(src, atype, dstIdSet, tLow, tHigh)
assoc_count(src, atype)
assoc_time_range(src, atype, tLow, tHigh, len)

"50 most recent check-ins to Golden Gate Bridge"

"10 most recent check-ins within last 24hr"
Architecture

Web servers

Cache

Database

Adapted from Bronson et al., ATC 13
Architecture

Web servers

Cache

Database

sharded by nodeID

Adapted from Bronson et al., ATC 13
Architecture

Web servers

Cache
“tier”

Database
sharded by nodeID

Adapted from Bronson et al., ATC 13
Architecture

Web servers

Cache

“tier”

Database

sharded by nodeID

objects, assoc lists, counts

Adapted from Bronson et al., ATC 13
Challenge: read load is too high
Challenge: read load is too high

Add more servers to the **caching layer**
Challenge: read load is too high

Add more servers to the caching layer

Challenge: graph grows larger
Challenge: read load is too high

Add more servers to the caching layer

Challenge: graph grows larger

Add more database shards to the storage layer
Challenge: read load is too high

Add more servers to the **caching layer**

Challenge: graph grows larger

Add more database shards to the **storage layer**

Challenge: a large tier of cache servers doesn’t scale well
Challenge: read load is too high

Add more servers to the **caching layer**

Challenge: graph grows larger

Add more database shards to the **storage layer**

Challenge: a large tier of cache servers doesn’t scale well

**Two-layer hierarchical caching**
Two-layer caching

Adapted from Bronson et al., ATC 13
Availability

- **Key idea:** a “tier” covers all ID space, can answer any query

- **Follower failure:** failover to another follower tier

- **Leader failure:** follower talks directly to database
  - 0.15% of follower cache misses

- **Database failure:**
  - If DB in master “region” down, promote a slave
  - 0.25% of a 90-day sample
  - If slave DB down: route to master
Write Path

- On write to node:
  - leader sends invalidate message to other followers

- On write to edge:
  - leader sends refill message (why?)

- More complicated when inter-region repl. is involved (see Figure)
Consistency

- As a whole, TAO is eventually consistent
- Within a tier, read-after-write consistency
- Trick: route critical queries to master region for strong consistency

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.
But, with failures, if client writes $N$ things...

- As a whole, TAO is eventually consistent
- Within a tier, read-after-write consistency
- Trick: route critical queries to master region for strong consistency

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.
Consistency

- As a whole, TAO is eventually consistent
- Within a tier, read-after-write consistency
- Trick: route critical queries to master region for strong consistency

But, with failures, if client writes N things…

Can end up with $2^N$ states!
Eval. Takeaway: API Frequency

Reads (99.8%)
- 40.9% assoc_range(src, atype, off, len)
- 28.9% obj_get(nodeId)
- 15.7% assoc_get(src, atype, dstIdSet, tLow, tHigh)
- 11.7% assoc_count(src, atype)
- 2.8% assoc_time_range(src, atype, tLow, tHigh, len)

Writes (0.2%)
- 52.5% assoc_add
- 20.7% obj_update
- 16.5% obj_add
- 8.3% assoc_del
- 2.0% obj_del
- 0.9% assoc_change_type
Eval. Takeaway: Degree

Takeaways:
1% supernodes long tail

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

• TAO uses a relational storage backend, citing operational confidence
  • Is a mature, full-fledged, performant, geographically distributed native graph store possible / preferable over TAO’s architecture?
  • Is there something fundamentally difficult/different about the higher-level data model that prevents this (vs. relational)?

• Is it possible to combine batch processing with online serving in a single graph system?

• Limitation: is stronger consistency worth the tradeoff in online graph serving?