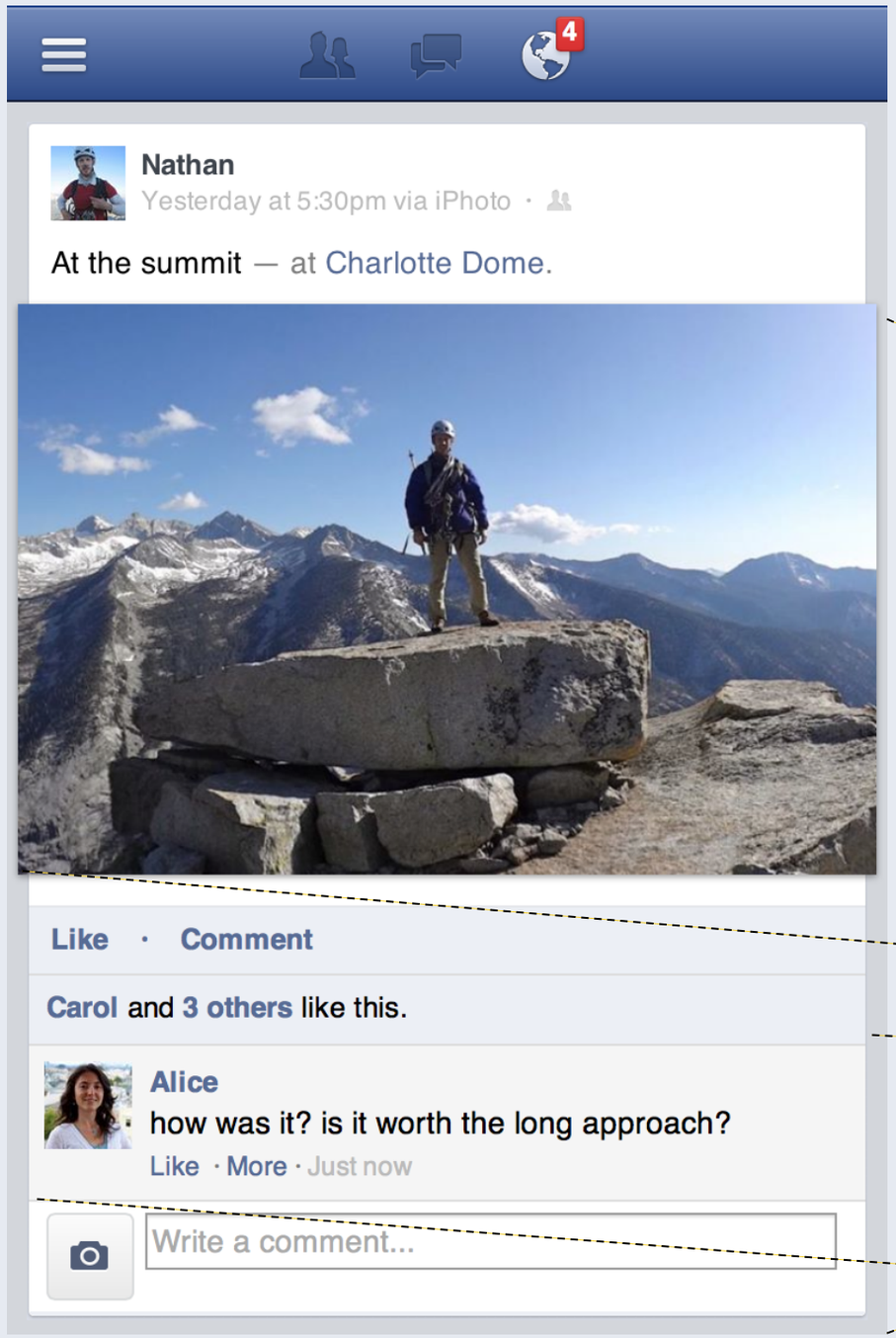
facebook

TAO
Facebook's Online Graph Store

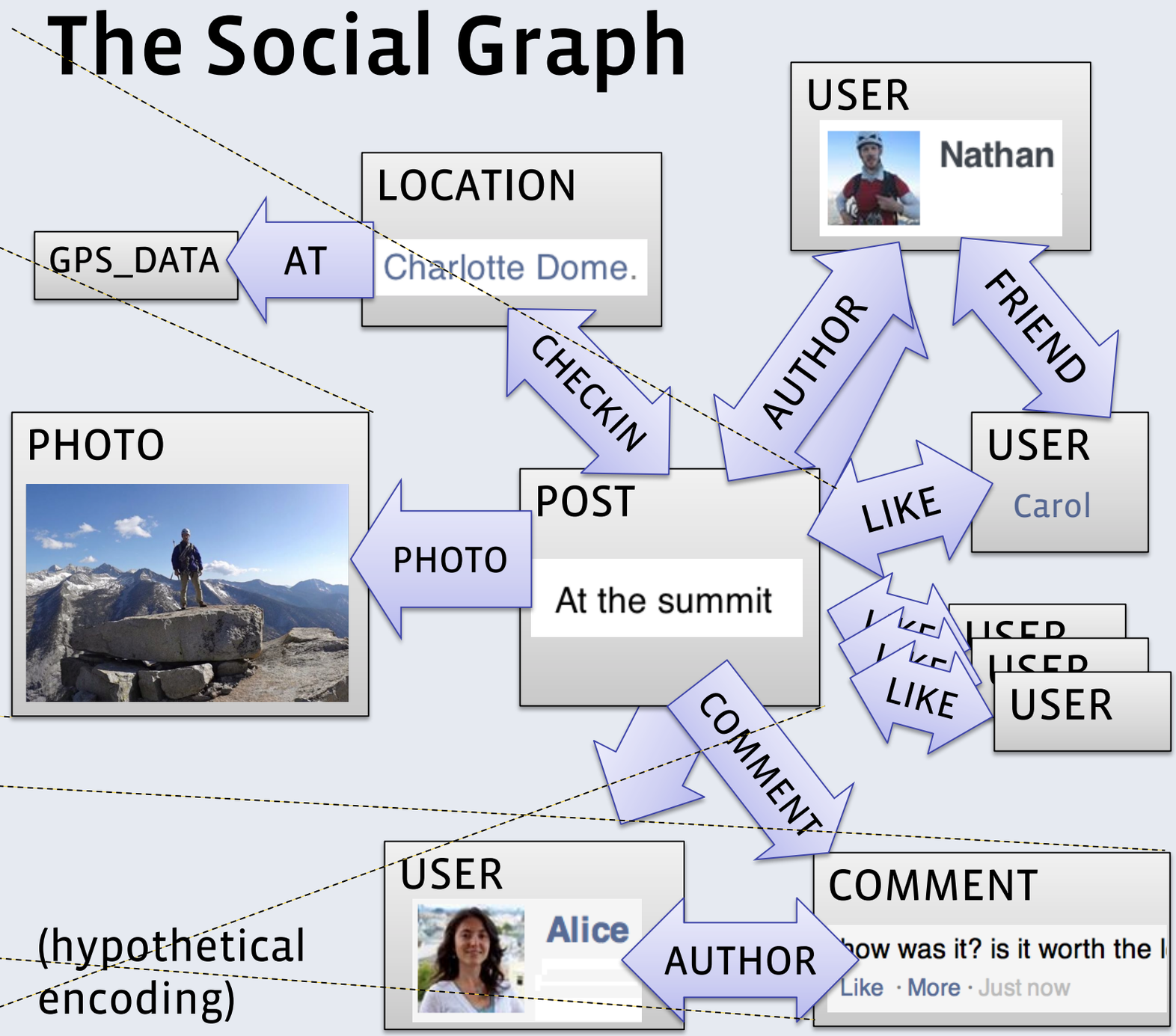
Nathan Bronson

CS244b- 11 Nov 2014

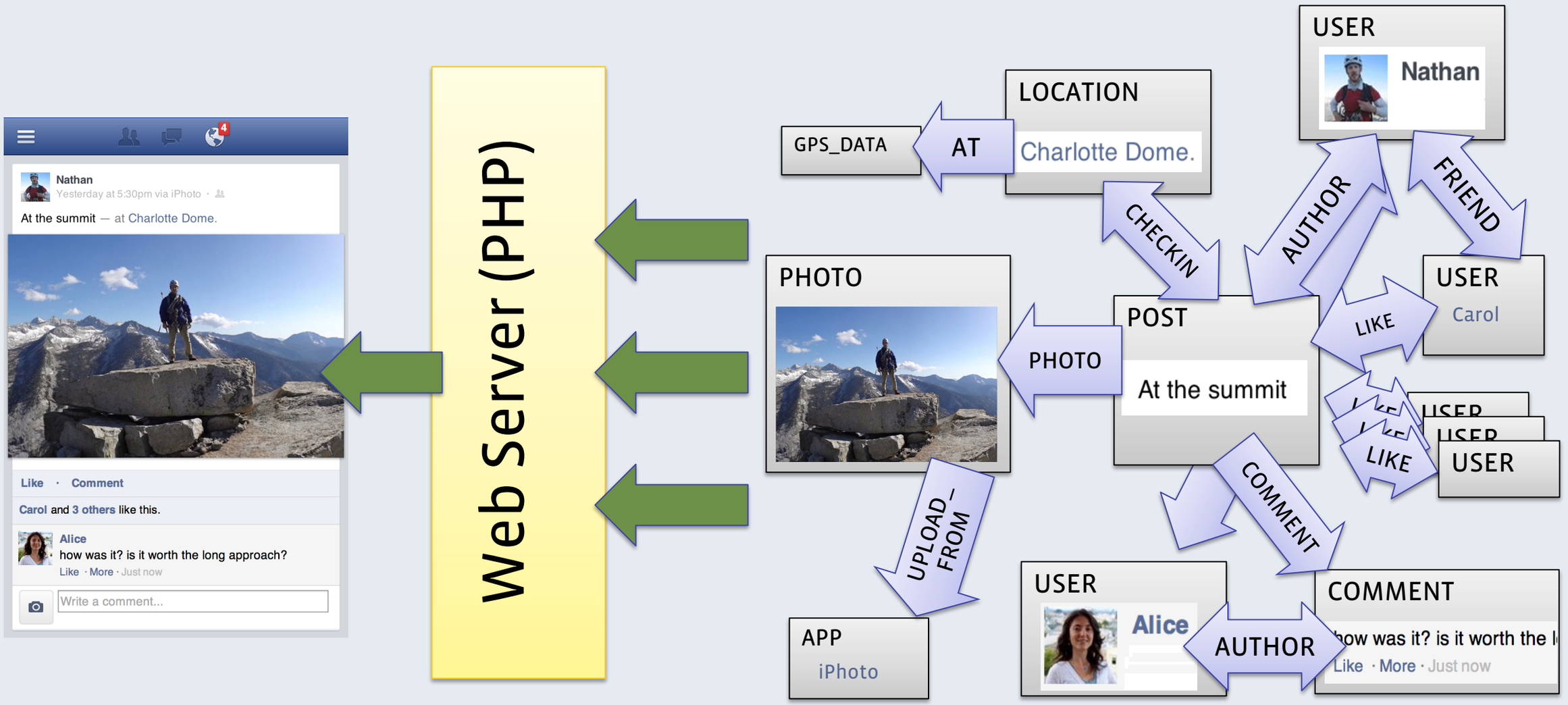
The Social Graph



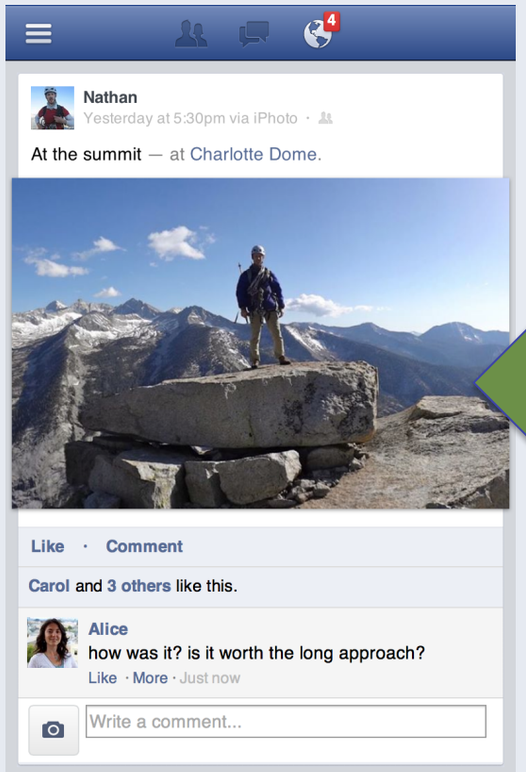
(hypothetical encoding)



Dynamically Rendering the Graph

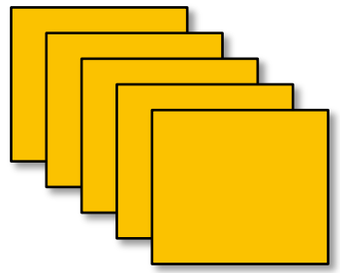


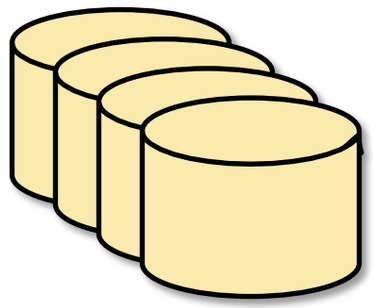
Dynamically Rendering the Graph

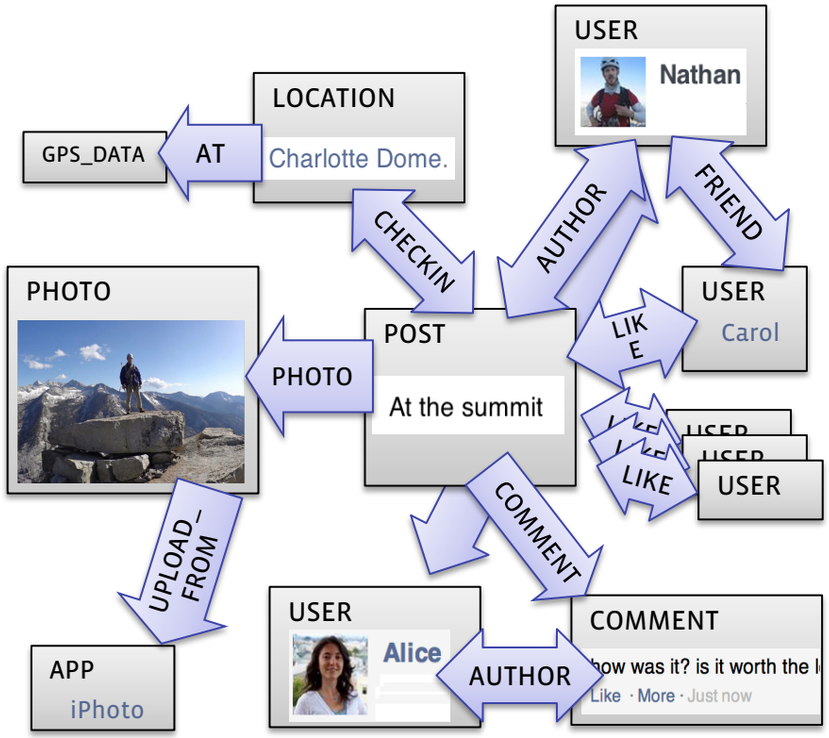


Web Server (PHP)

TAO







- >1 billion queries/second
- many petabytes of data

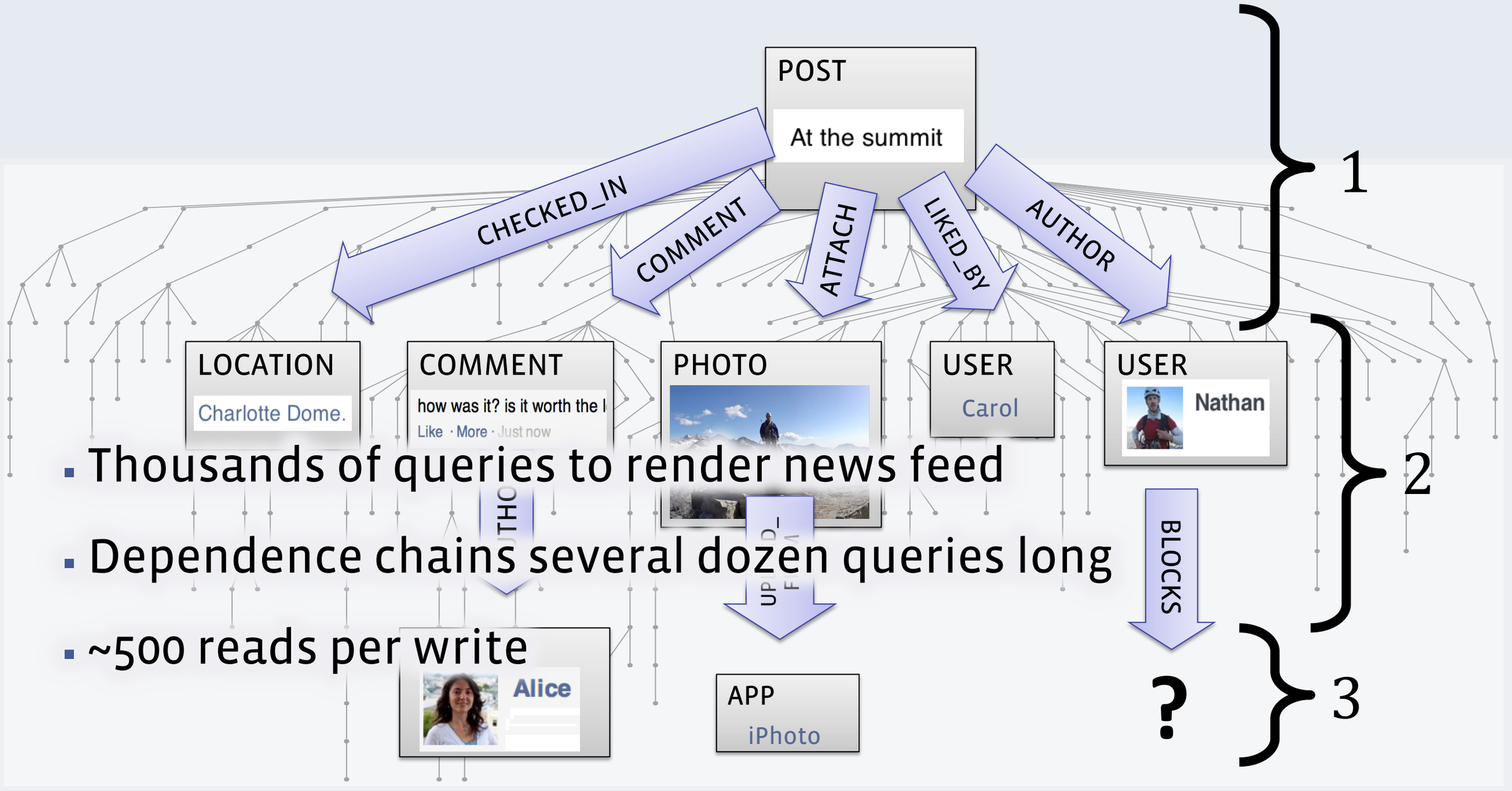
What Are TAO's Goals/Challenges?

- Efficiency at scale

facebook

December 2010

Data Dependencies and Query Rounds

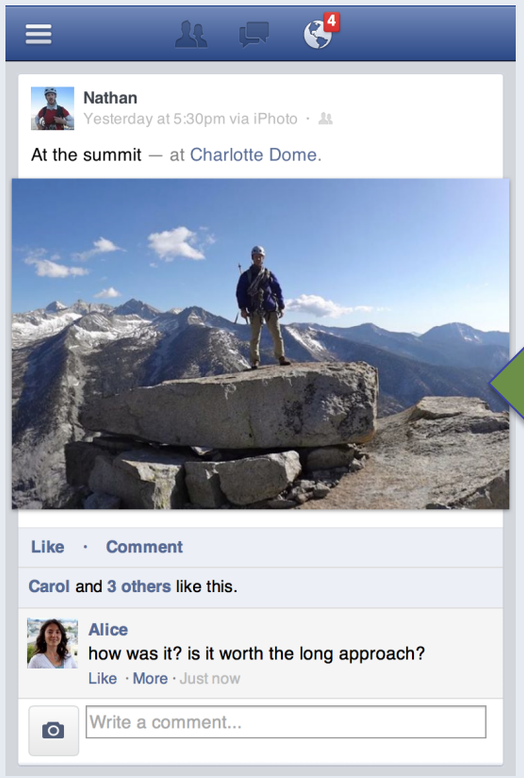


- Thousands of queries to render news feed
- Dependence chains several dozen queries long
- ~500 reads per write

What Are TAO's Goals/Challenges?

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

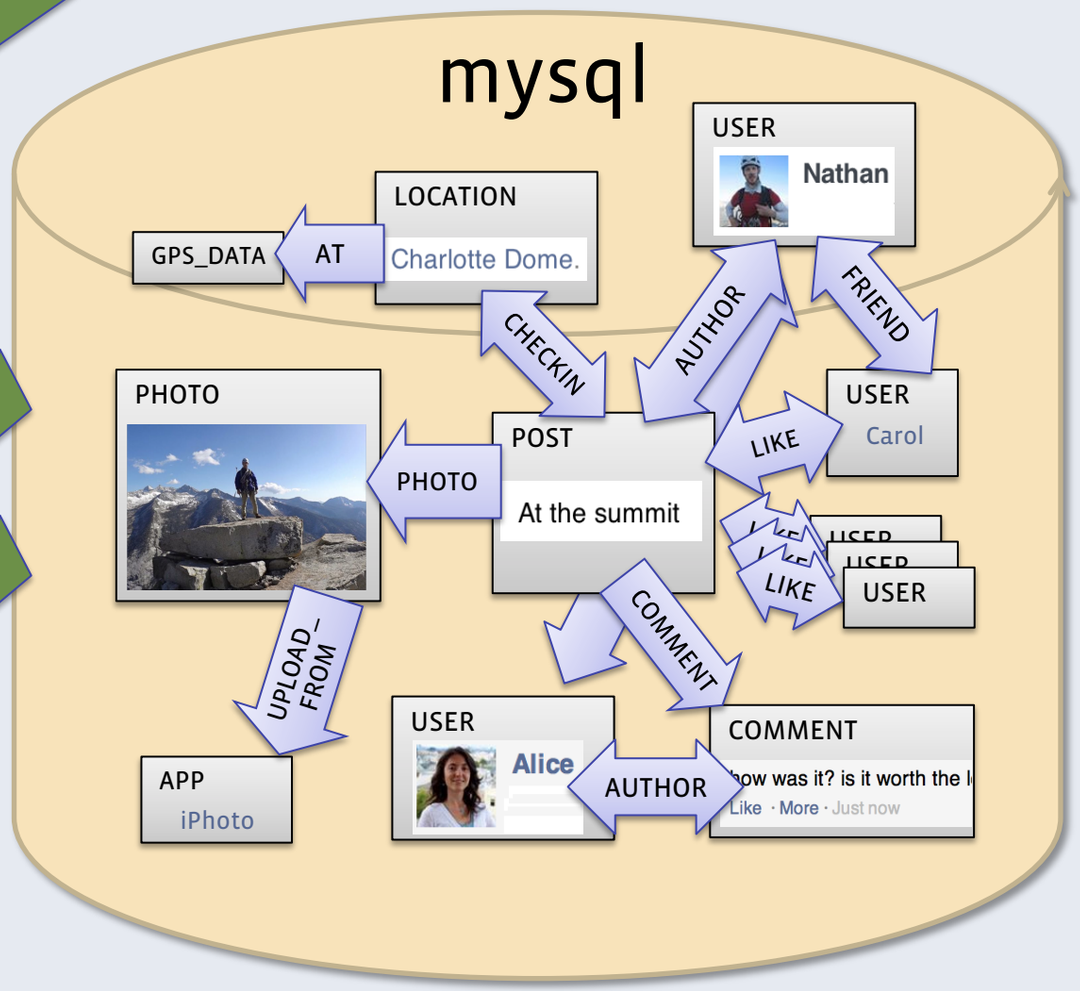
Graph in Memcache



Web Server (PHP)

Obj & Assoc API

memcache
(nodes, edges, edge lists)

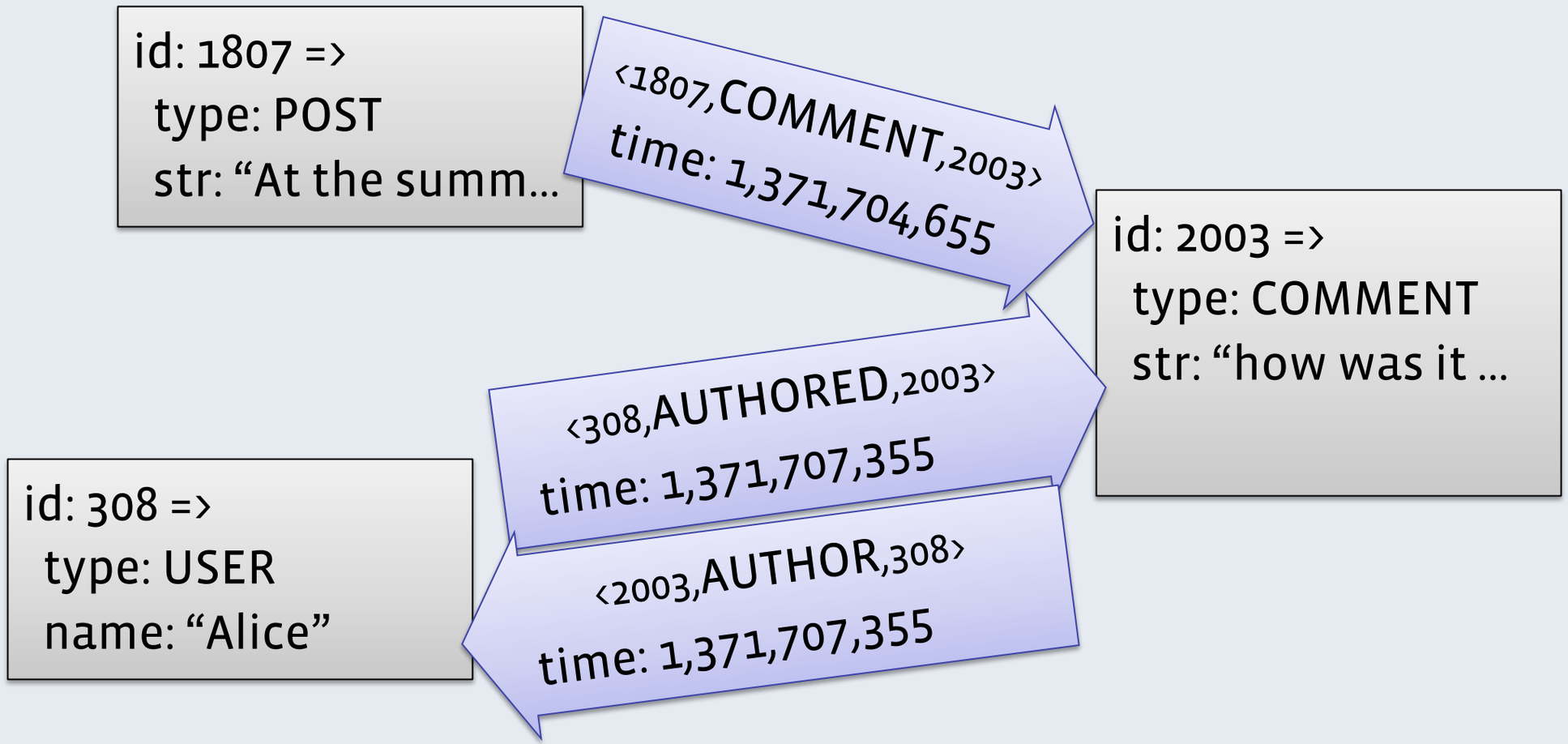


Objects = Nodes

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

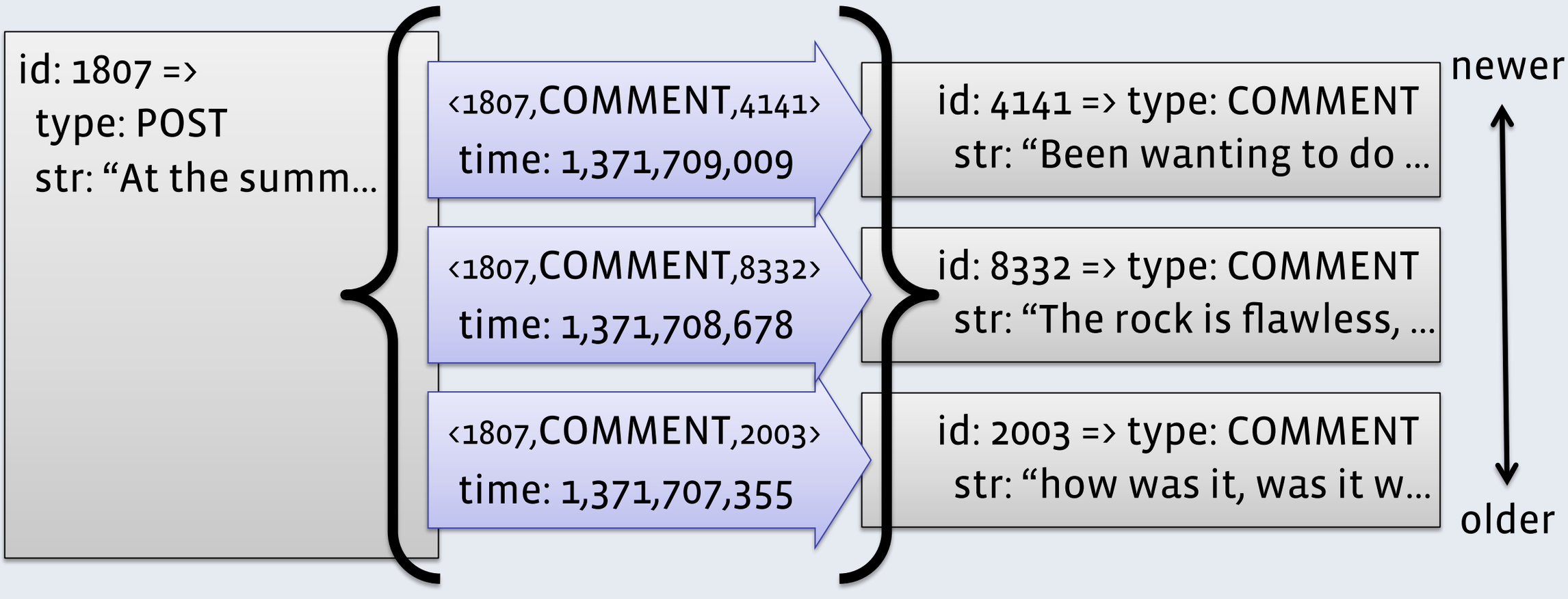
Associations = Edges

- Identified by <id1, type, id2>
- Bidirectional associations are two edges, same or different type



Association Lists

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



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

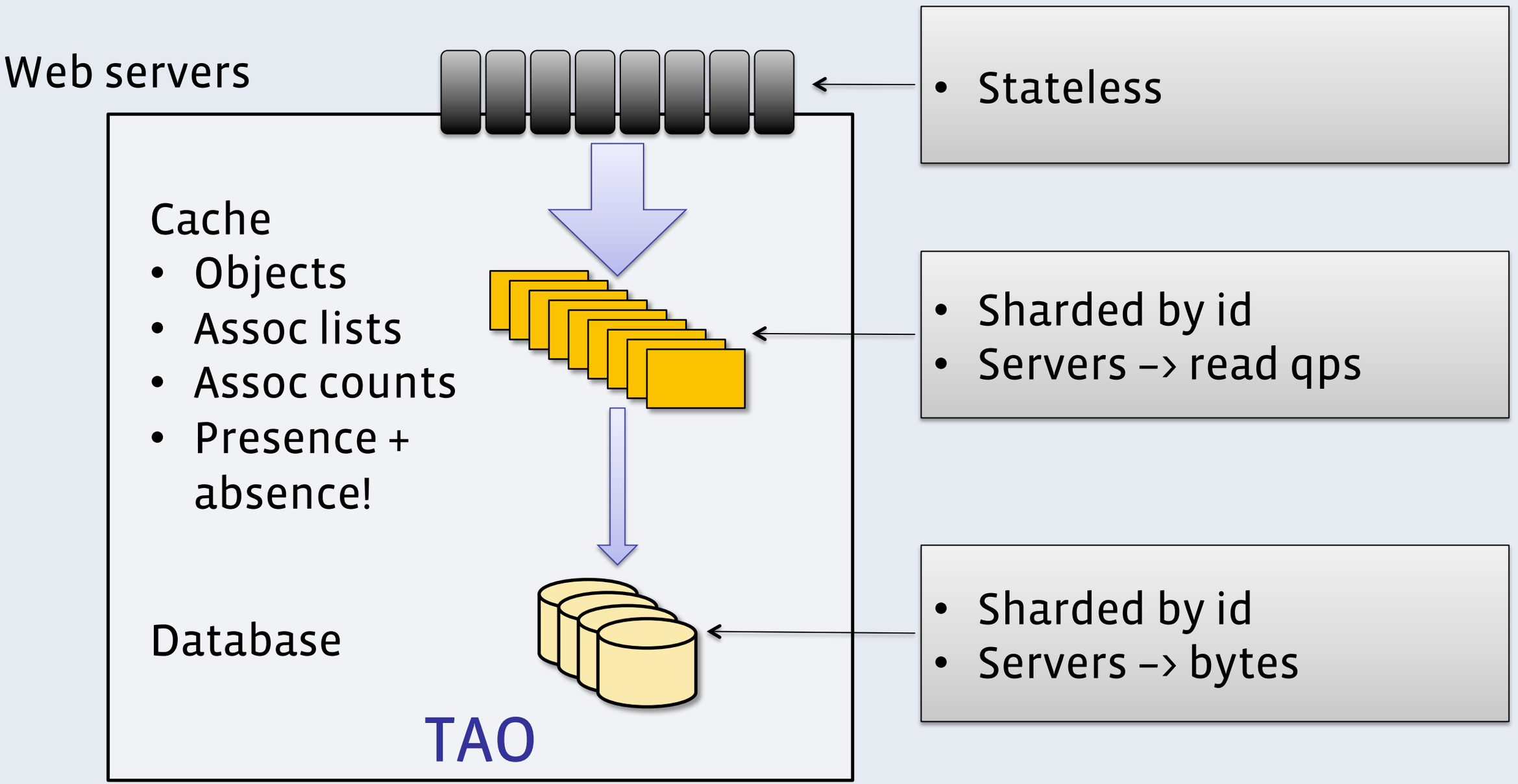
facebook

December 2010

Mapping FBIDs to machines

- Two-level mapping
  - Fixed n-to-1 mapping from fbid -> shard
  - Dynamic n-to-1 mapping from shard -> machine
- Advantages:
  - Efficient and scalable
  - Allows good load balancing
  - Allows explicit colocation or anti-colocation
- Disadvantages:
  - Colocation is forever

Independent Scaling by Separating Roles

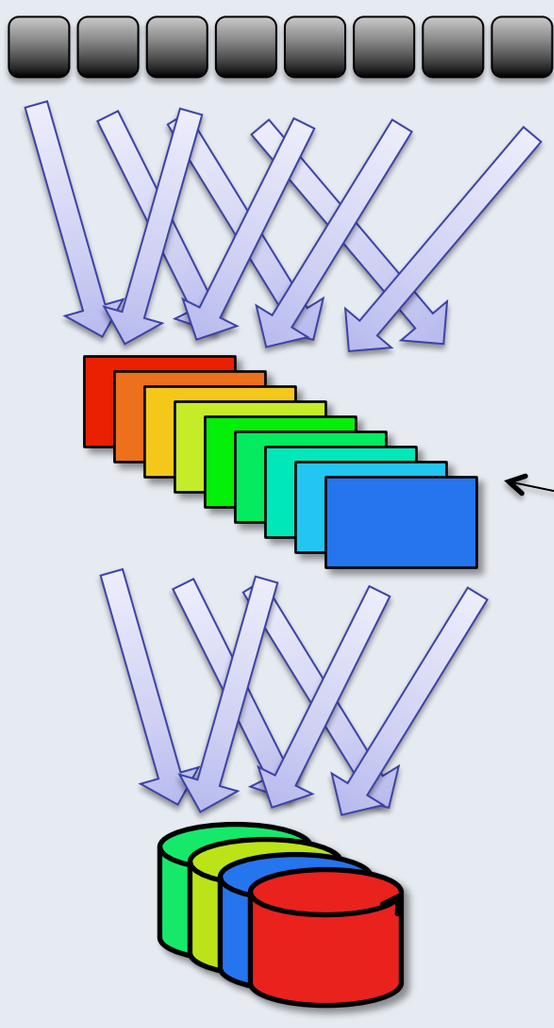


Subdividing the Data Center

Web servers

Cache

Database



- Inefficient failure detection
- Many switch traversals

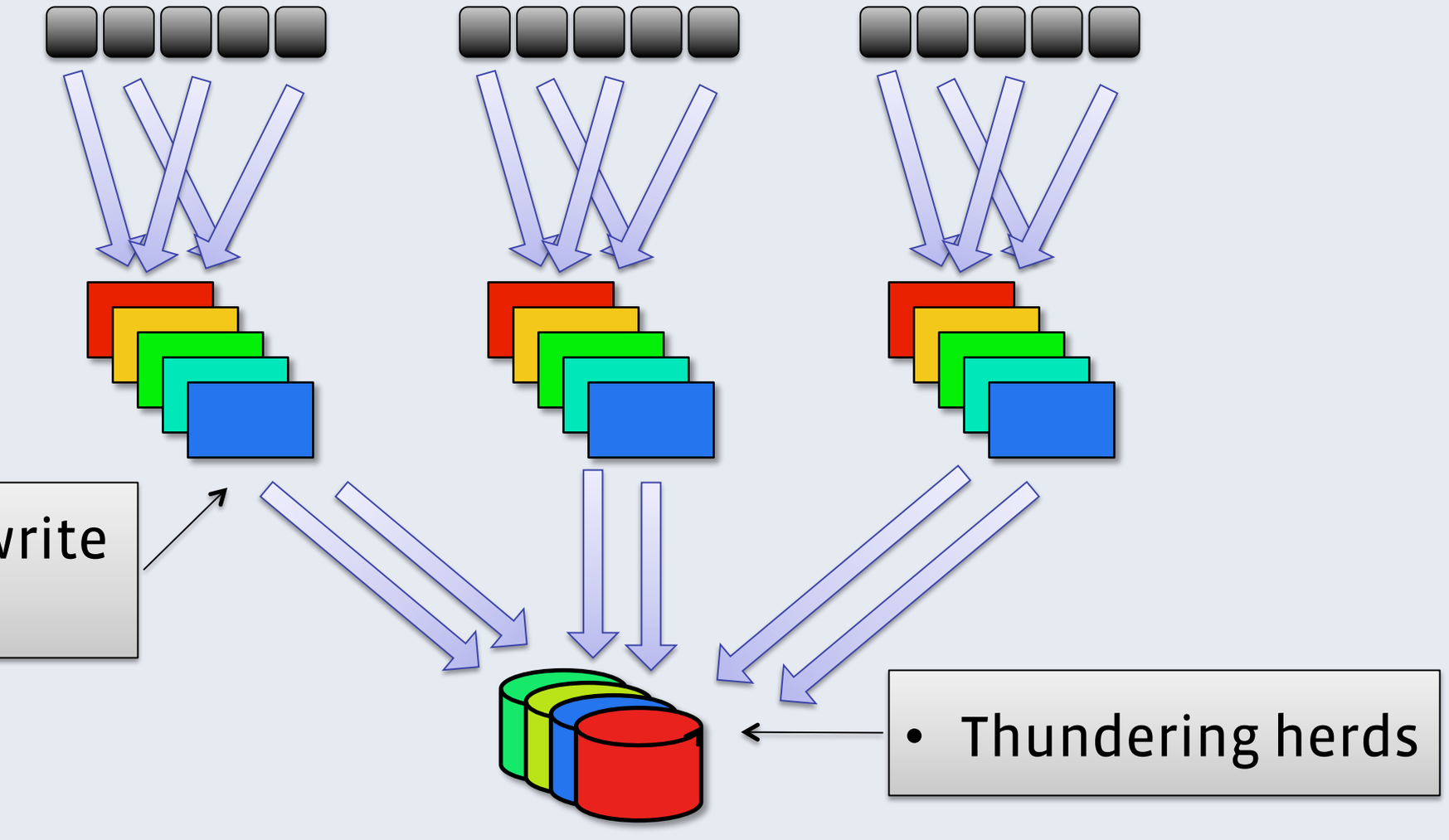
- Many open sockets
- Lots of hot spots

Subdividing the Data Center

Web servers

Cache

Database



- Distributed write control logic

- Thundering herds

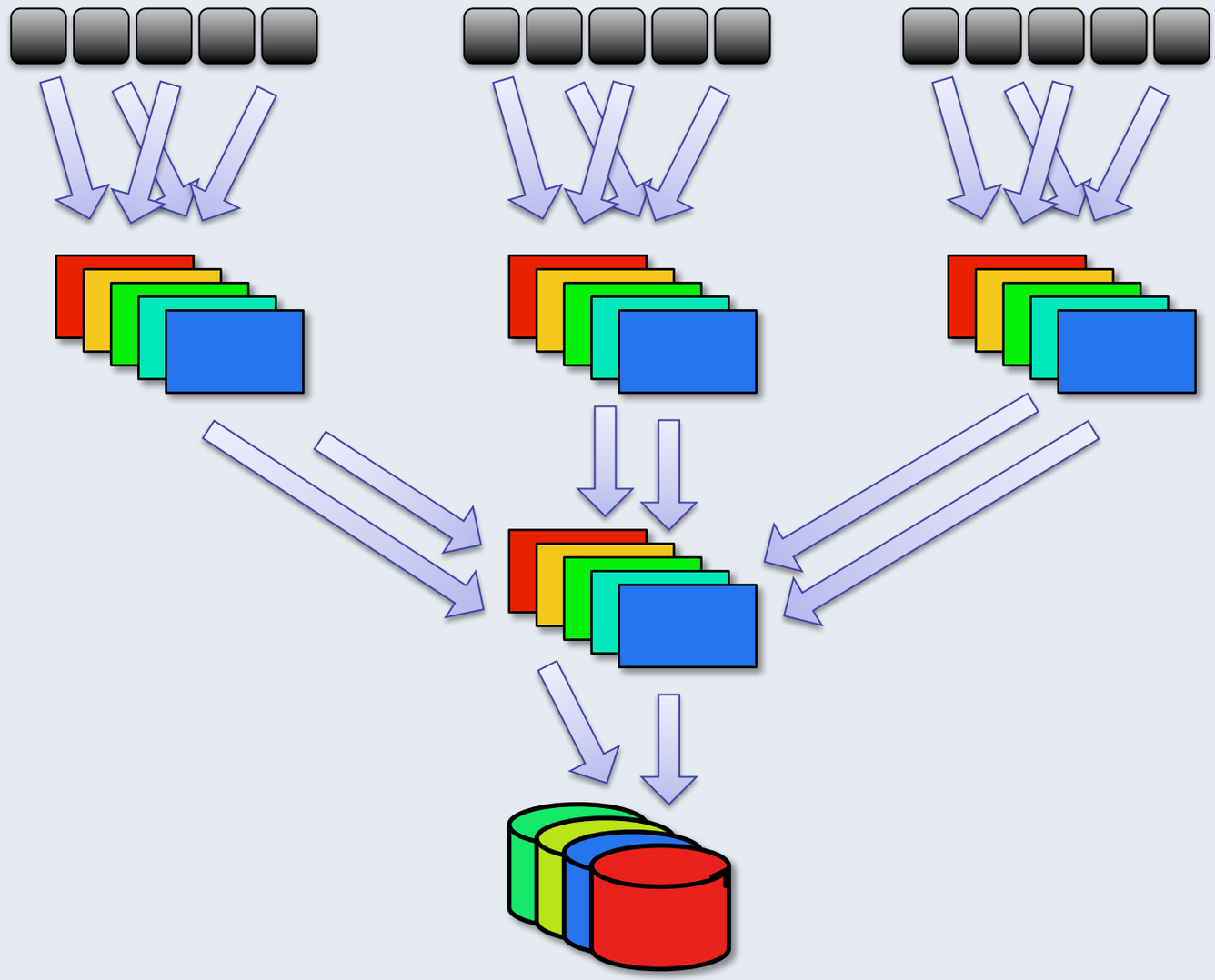
Follower and Leader Caches

Web servers

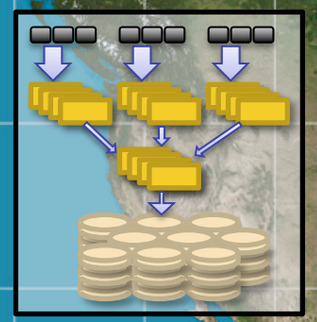
Follower cache

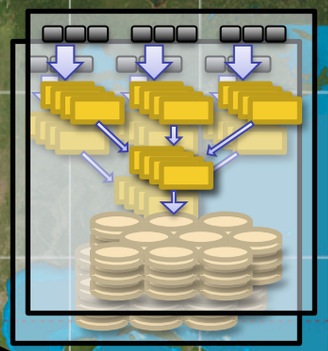
Leader cache

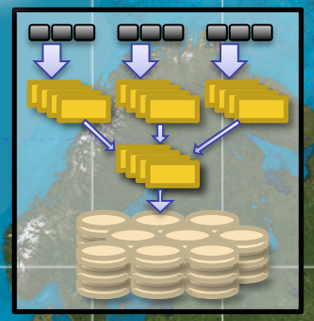
Database



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

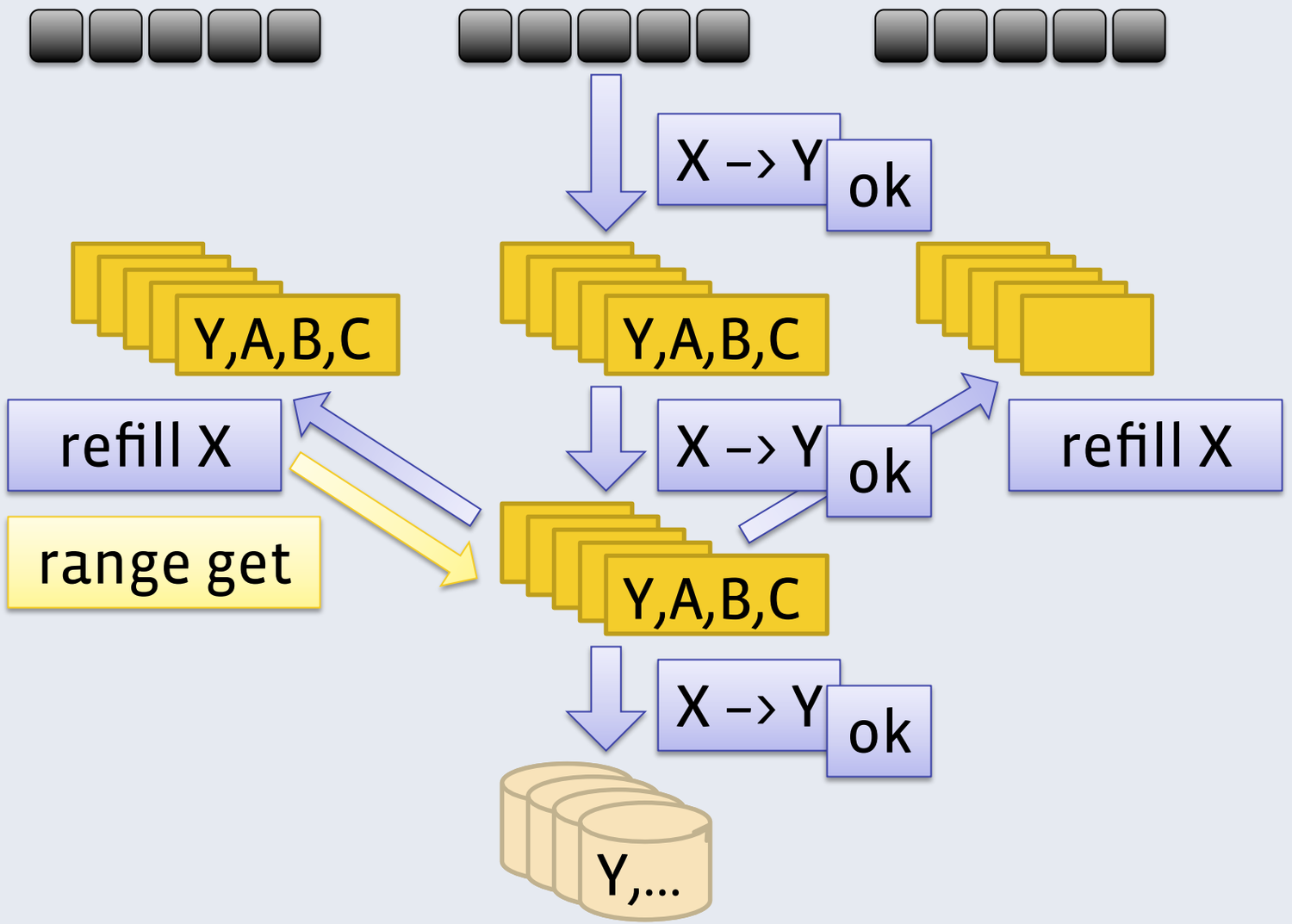
Write-through Caching – Association Lists

Web servers

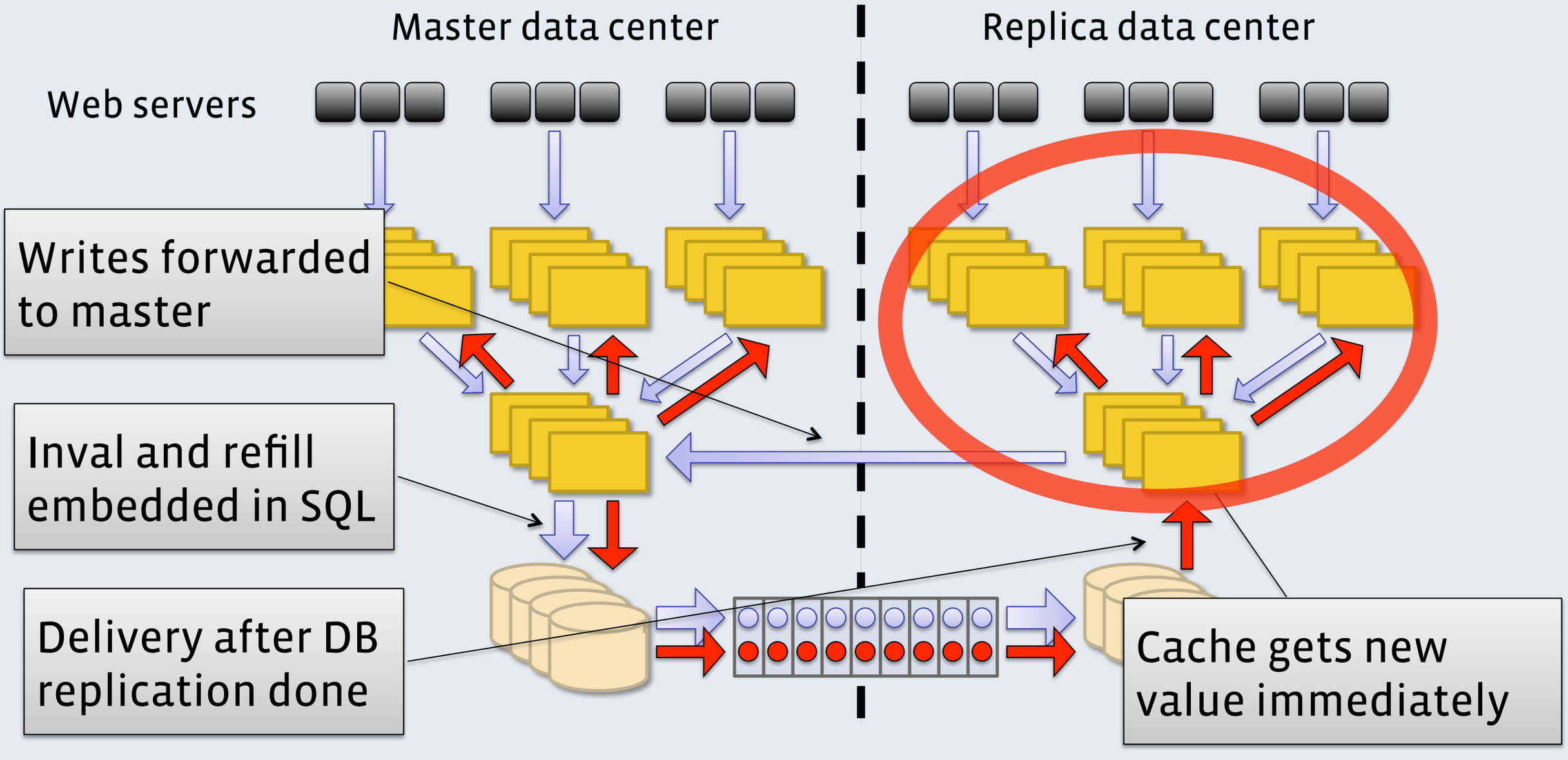
Follower cache

Leader cache

Database



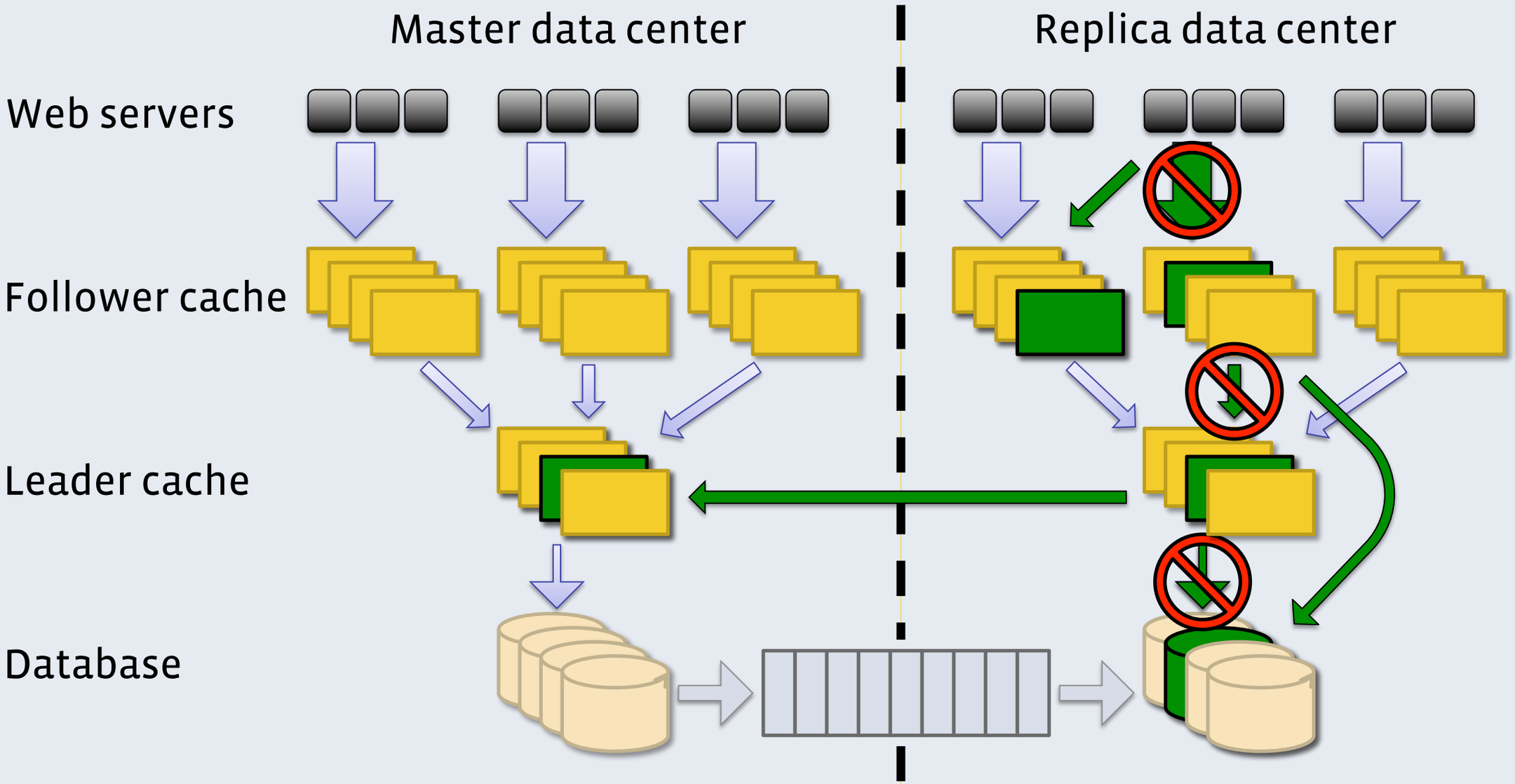
Asynchronous DB Replication + Local Read-What-You-Wrote



What Are TAO's Goals/Challenges?

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

Improving Availability: Read Failover



Move Fast

Efficiency at scale Read latency	• Separate cache and DB • Graph-specific caching • Efficient negative caches • Subdivide data centers
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^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?

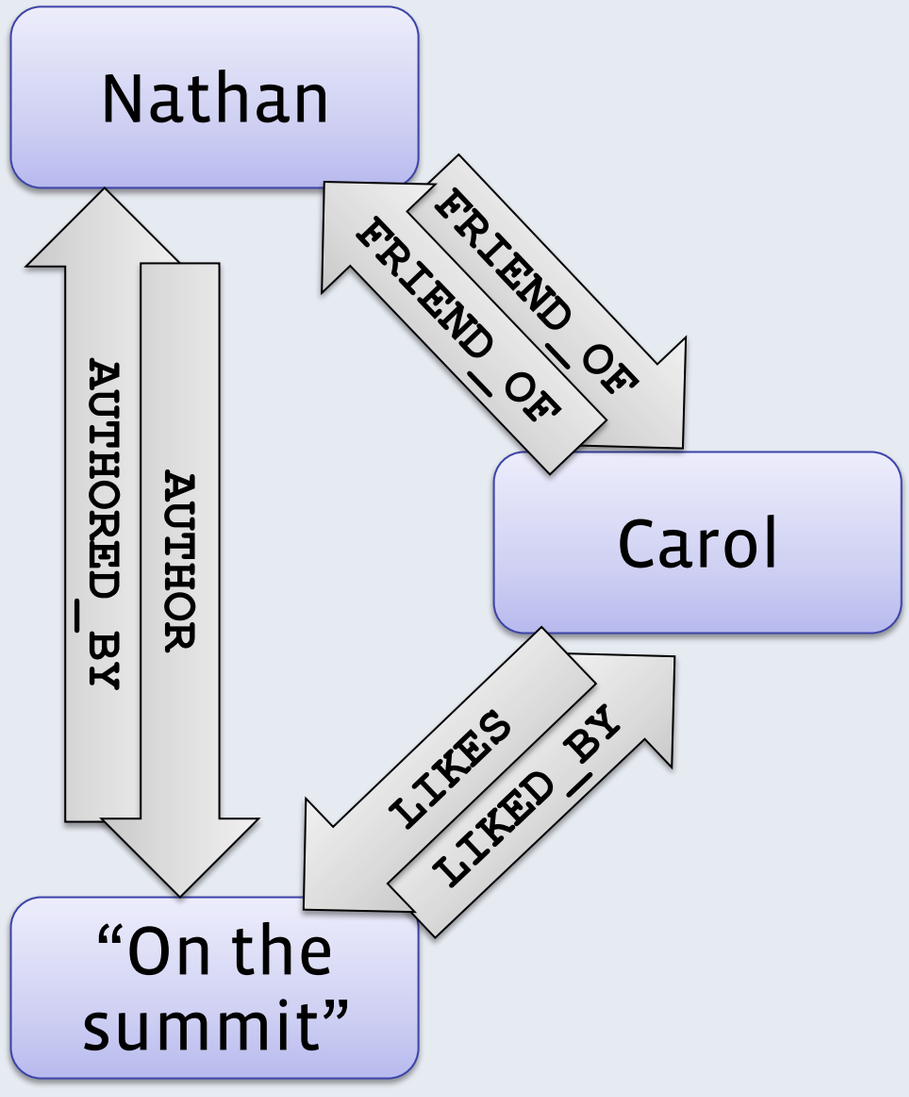
Questions?

Efficiency at scale Read latency	• Separate cache and DB • Graph-specific caching • Efficient negative caches • Subdivide data centers
Write timeliness	• Write-through cache • Asynchronous replication
Read availability	• Alternate data sources

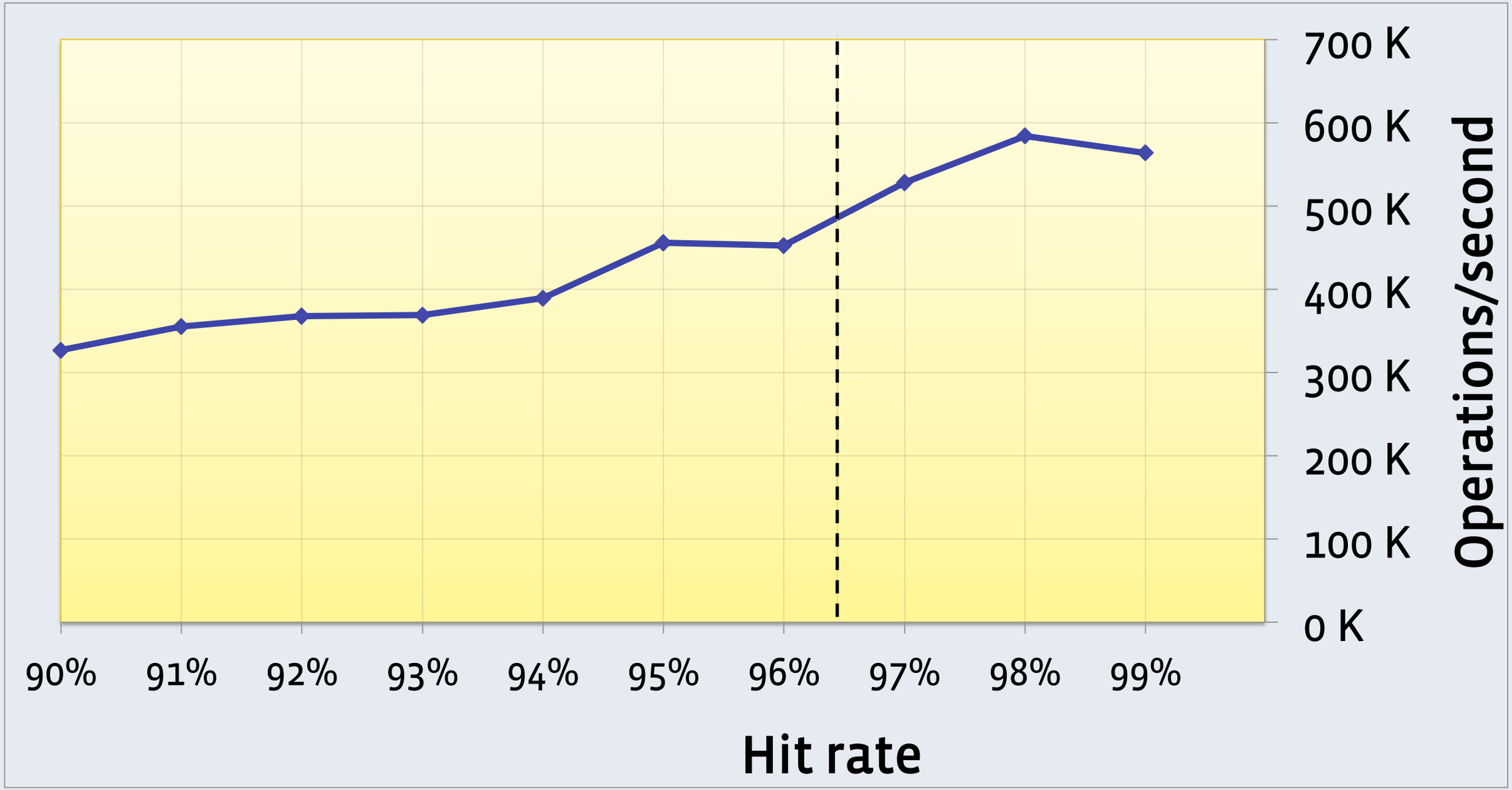
facebook

Inverse associations

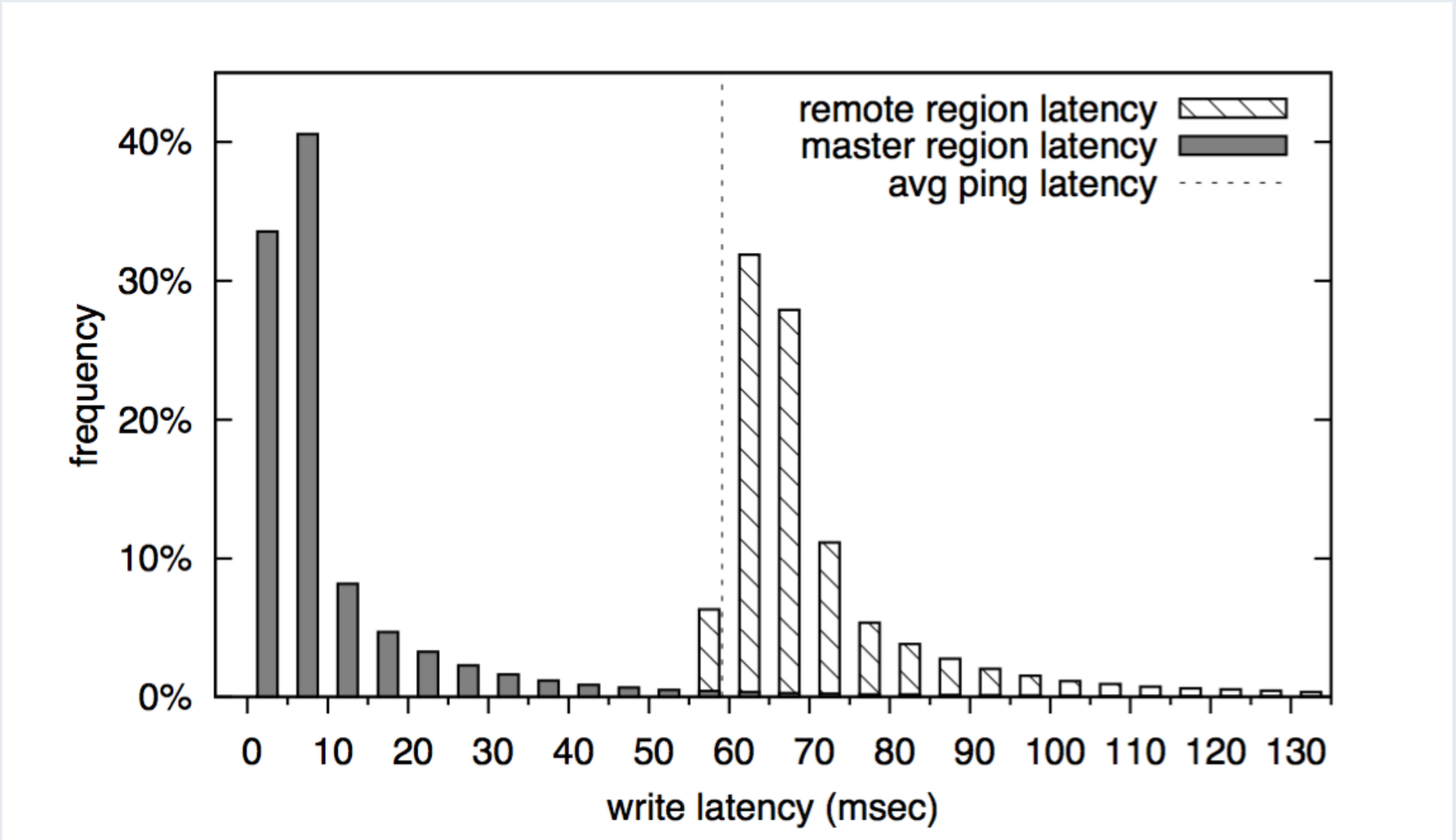
- Bidirectional relationships have separate *a*→*b* and *b*→*a* edges
  - inv_type(**`LIKES`**) = **`LIKED_BY`**
  - inv_type(**`FRIEND_OF`**) = **`FRIEND_OF`**
- Forward and inverse types linked only during write
  - TAO **`assoc_add`** will update both
  - Not atomic, but failures are logged and repaired



Single-server Peak Observed Capacity



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