Ge Song, Frédéric Magoulès (Supervisor), Fabrice Huet (Co-Supervisor)
Lab MICS
CentraleSupélec
Université Paris-Saclay

Parallel and Continuous Join Processing for Data Stream

Thèse pour l'obtention du grade de Docteur

Table of Contents

Background


BIG
DATA
INFORMATION
PETABYTES
SEARCH
ANALYSIS
RESEARCH
STORAGE
LARGE
SETS
MILLION
SOFTWARE
SAN
NAS
SOCIAL
YEARS
RESULTS
DRIVE
REPLICATION
HANDLE
SOURCES
LIMITS
CHALLENGES
COMPLEX
ANNOUNCED
PAST
HAND
CAPACITY
INCLUDE
NETWORKS
HYPOTHESIS
SCIENCE
END
MOST
NEARLY
MAPREDUCE
FUTURE
CRITIQUES
GOVERNMENT
FACTOR
SYSTEMS
MANAGE
ONLINE
DAY
WORLDWIDE
PROCESS
MAKING
INTEGRATION
VOLUME
TECHNOLOGIES
ONE
OUT
DEVELOPMENT
TECHNOLOGY
MANY
APPROACH
BILLION
NEW
PARALLEL
STATE
CRITIQUE
PEOPLE
TRAFFIC
USE
CURRENTLY
MASSIVE
DEFINITION
USING
FRAMEWORK
BETWEEN
STATISTICS
BUSINESS
EFFECTIVE
VISUALIZATION
DATABASES
SIZE
ANALYTICS
SIMULATIONS
PROCESSING
MANAGEMENT
DETERMINE
WELL
EXABYTES
SET
DEPARTMENT
HIGHER
ALGORITHMS
COMPANIES
QUERIES
HIGH
USERS
AMOUNT
RATE
REAL
STRUCTURE
WORK
APPLICATIONS
INFRASTRUCTURE
TERMS
YEAR
PRIVATE
CAPABILITIES
SUCCESSFUL
NEEDED
USES
LEARNING
LESS
HUGE
INTERNET
COMPARED
EVERY
WORLD
VARIOUS
BASED
TIMES
SHARED
ARCHITECTURE
PRIVACY
INTERNATIONAL
IMPLEMENTATION
BECOMES
SECOND
NATIONAL
INTELLIGENCE
FLOW
APPROACHES
PROCESSES
PARADIGM
FUNDING
RECORDS
COST
INSIGHT
TOOLS
CENTER
INITIATIVE
ECONOMIC
PROJECT
DISTRIBUTED
PER
RELEVANT
TERABYTES
MARKET
VARIETY
ANNUAL
USED
RELATED
TIME
SENSOR
WOULD
NOW

Part I: Data Driven Stream Join (kNN)

Outline

- Related Work
- Parallel Workflow
- Theoretical Analysis
- Continuous kNN
- Experiment Result
- Conclusion

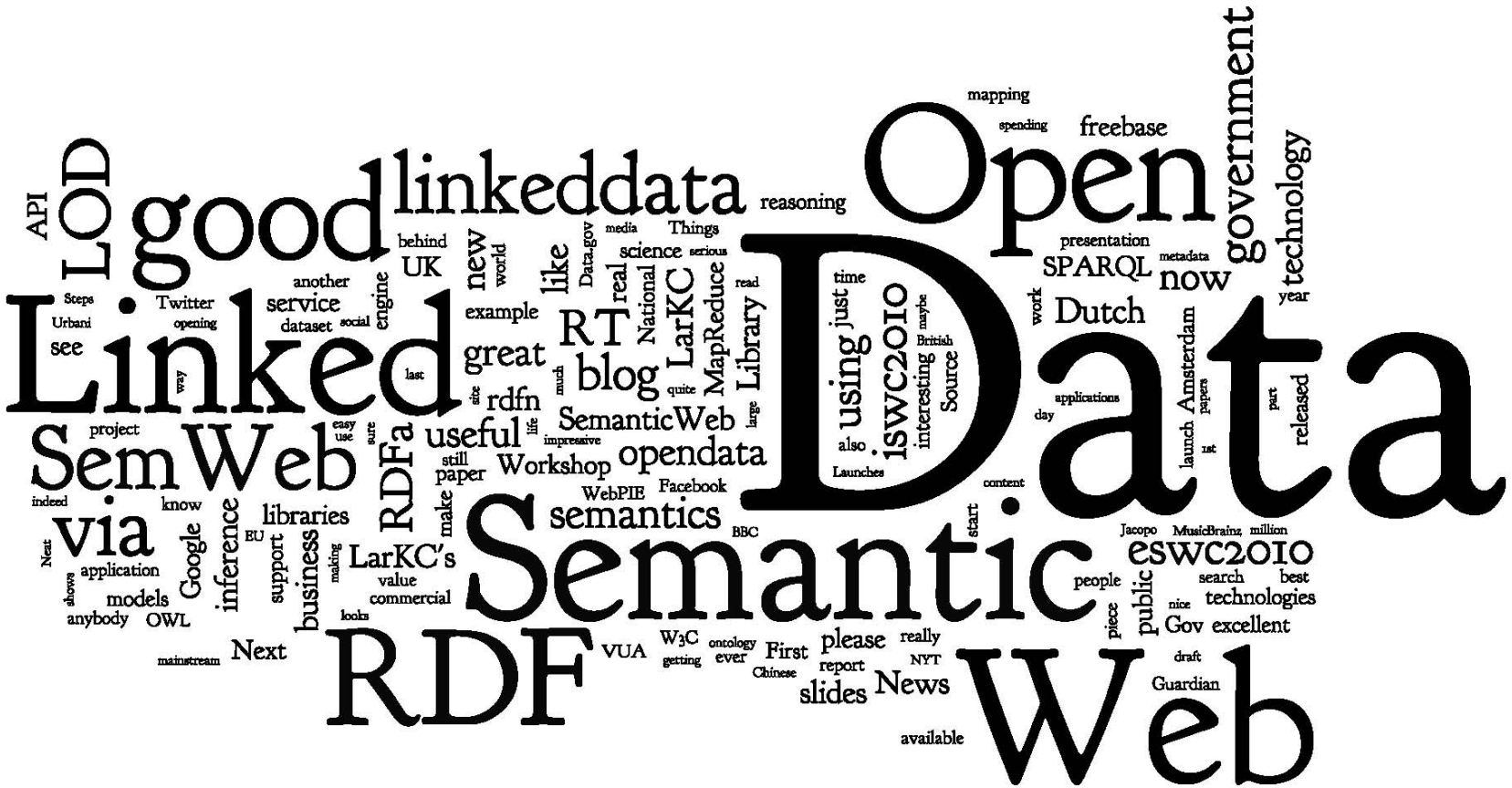
Outline

- Related Work
- Parallel Workflow
- Theoretical Analysis
- Continuous kNN
- Experiment Result
- Conclusion

Definition: kNN

Given a set of query points $\mathbb{R}$ and a set of reference points $\mathbb{S}$, a $\mathbb{k}$ **nearest neighbor join** is an operation which, for each point in $\mathbb{R}$, discovers the $\mathbb{k}$ nearest neighbors in $\mathbb{S}$.

Part II: Query Driven Stream Join (RDF)



Outline

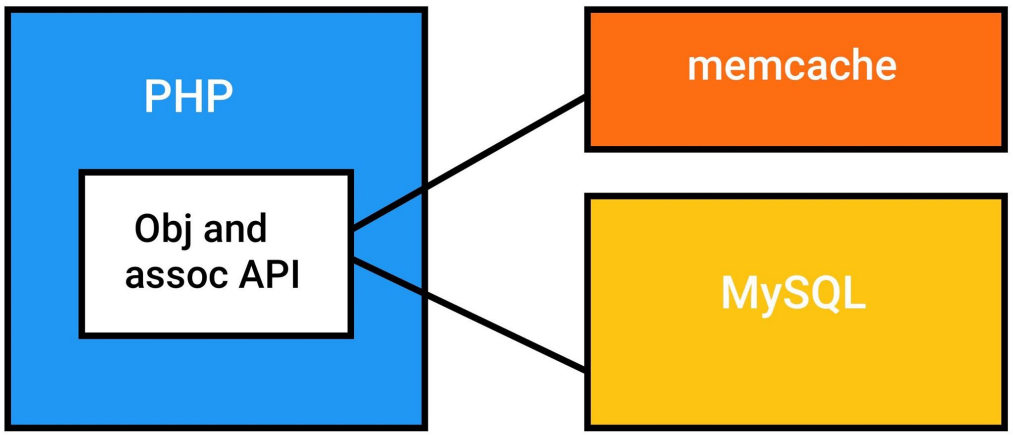
- Related Work
- Query Decomposition and Distribution
- Data Partition and Assignment
- Parallel and Distributed Query Plan
- Continuous Join
- Analysis
- Implementation
- Experiment Result
- Conclusion

Outline

- Related Work
- Query Decomposition and Distribution
- Data Partition and Assignment
- Parallel and Distributed Query Planner
- Continuous Join
- Analysis
- Implementation
- Experiment Result
- Conclusion

Architecture

Before Tao

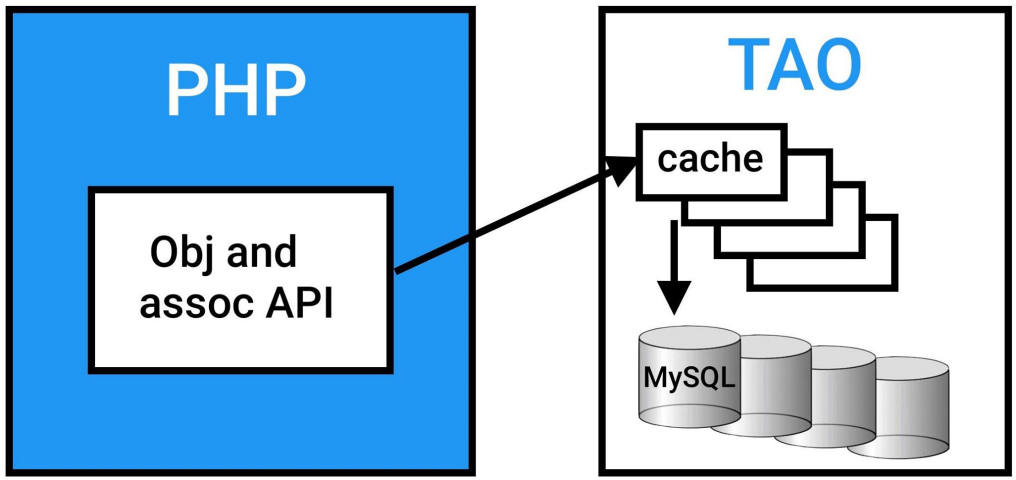


After Tao

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 end 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

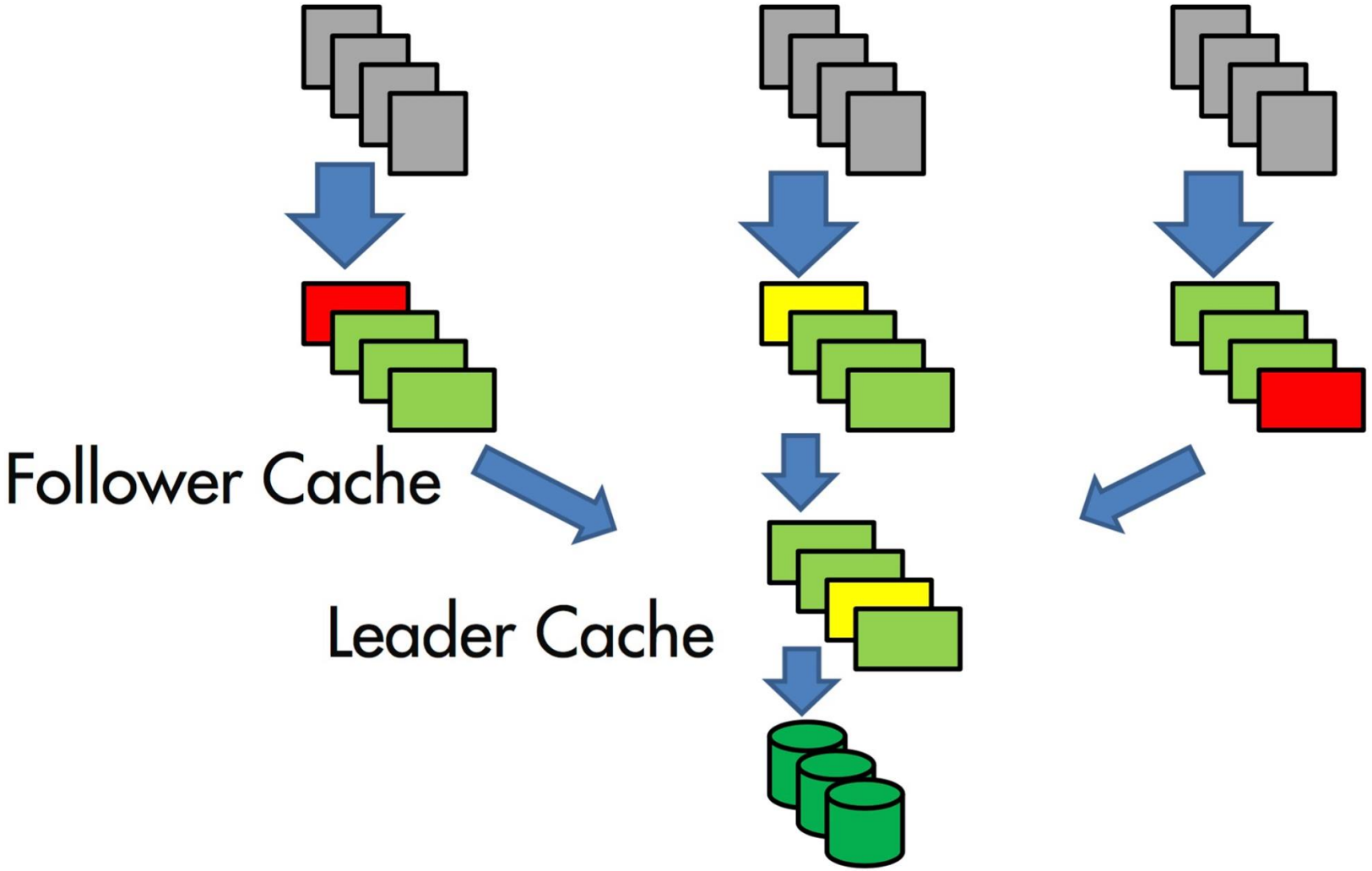
Yet Another caching layer

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

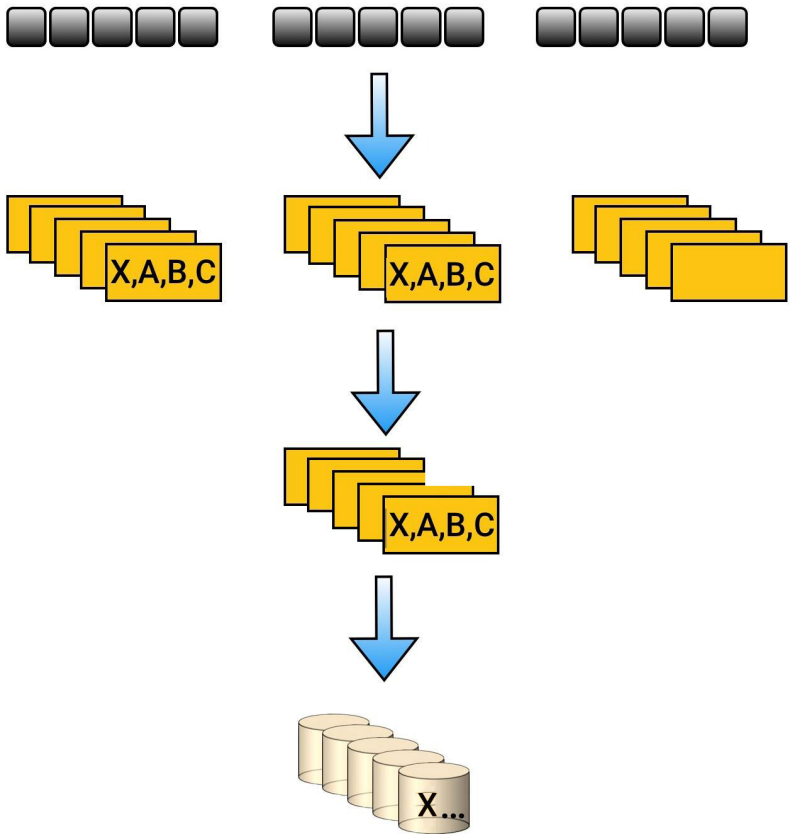
Leaders & Followers



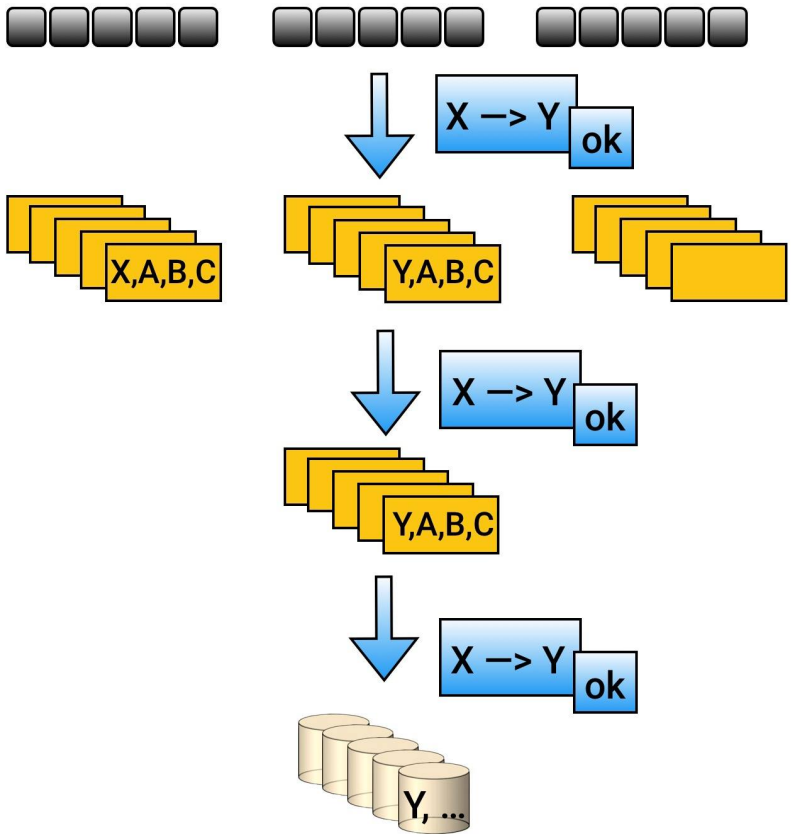
Leaders & Followers

- 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

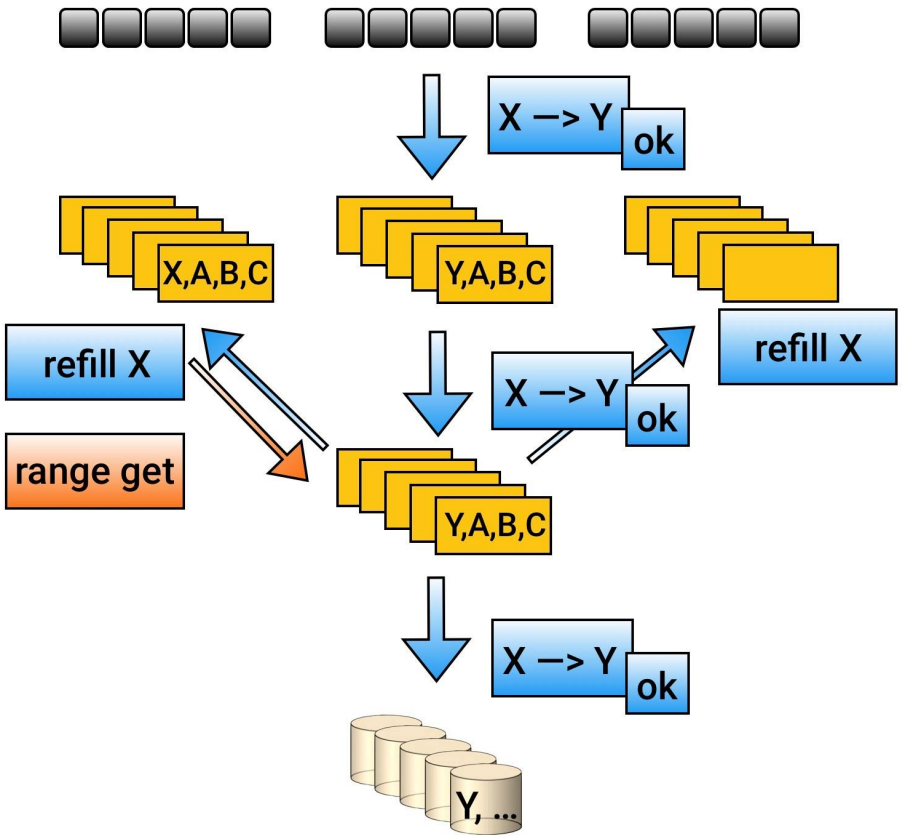
Leaders & Followers



Leaders & Followers



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

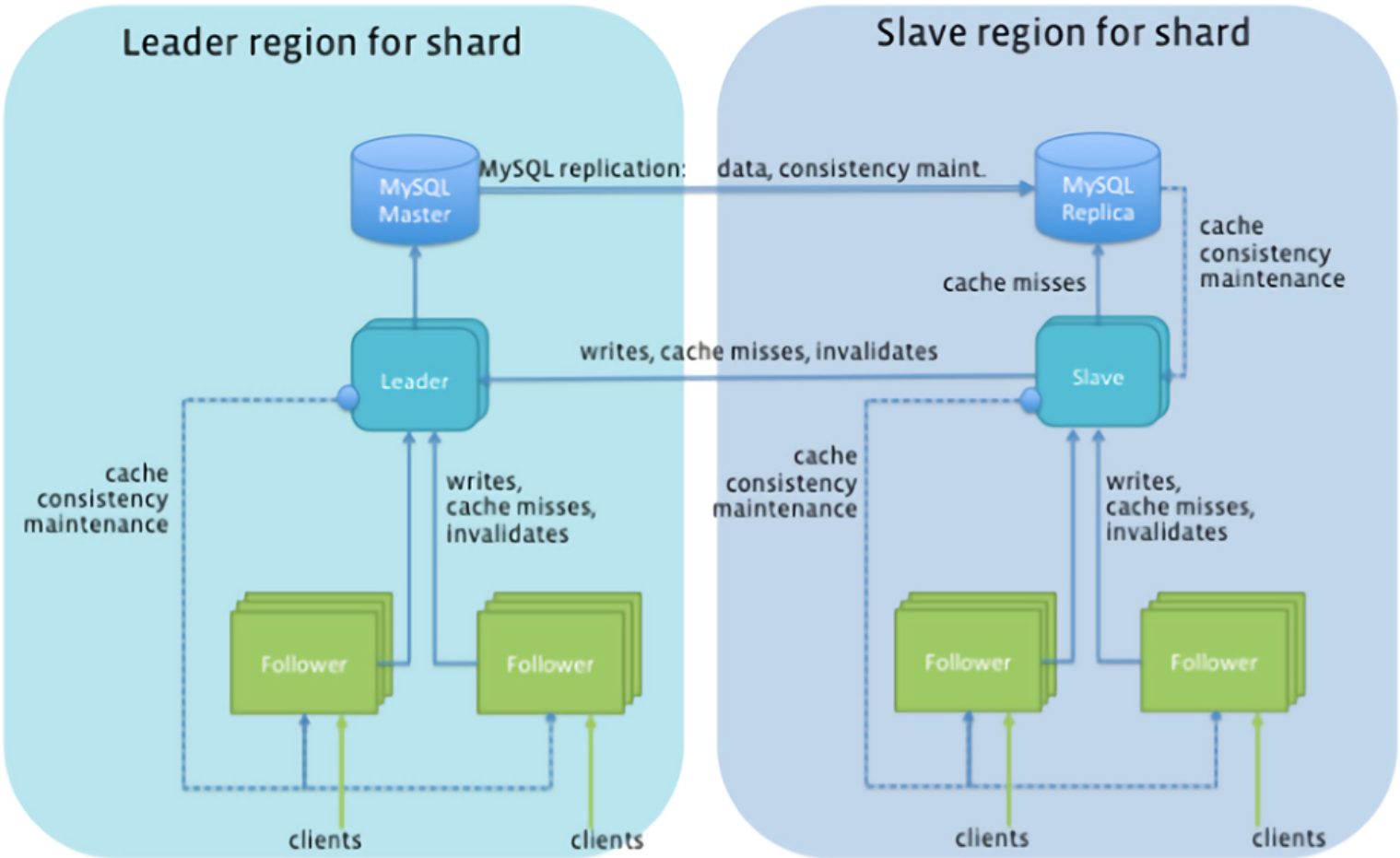
Solution: Handles read cache miss locally

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
- Cache invalidation message are embedded into db replication stream
- Slave leader will update it’s cache as soon as write are forwarded to master

Overall Architecture



Implementation

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

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
  - stored separately
  - 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 chache 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

Thank You!