# Twitter workload for NoSQL databases

University of Minho

# Context

- Social networks applications have taken a big growth.
- MySpace, Facebook, Twitter, Hi5, Orkut, Bebo, LinkedIn, PatientsLikeMe, Yahoo!360.
- Are in top of the sites with more traffic and have millions of users worldwide.

# Context

- User's actions and preferences may affect many users in their network.
- Pose new challenges to current database servers.
- Use of centralised RDBMS or even a replicated DBMS is a major bottleneck.
- Social applications are thus exploiting NoSQL databases.

# Context

- No benchmarks mimicking the workload of a social network.
- Existent NoSQL benchmarks are naive.
- Standard benchmarks (like TPC-C,...) not suited for large scale storage system.

# Goals

- Create a benchmark based on a twitter alike application:
  - Measure performance (throughput, latency, ...).
  - Behaviour of databases in face of faults and scalability
  - Workload to simulate as close to real the use of the application.
- Event based API that allows to evaluate real and simulated NoSQL databases.
- Architecture compatible with Cloud Environment.

# Workload entities

## Users

| key | value |
|---|---|
| userID | name, password, creation date, followers, following and lastTweetID |
| ... | ... |

## Tweets

| key | value |
|---|---|
| userID-tweetID | tweet |
| ... | ... |

## FriendsTimeLine

| key | value |
|---|---|
| userID | List<date:tweetID> |
| ... | ... |

## Tags

| key | value |
|---|---|
| tag | List<tweets> |
| ... | ... |

# Workload initialisation

- Social graph
  - small-world network (due to high clustering and small diameter).
  - power-law distribution (few nodes have high degree, while the majority of nodes have small degree).
  - scale-free.
- Initial tweets per user.

# Workload ops

- statuses_user_timeline (userID)
- statuses_friends_timeline (userID)
- statuses_mentions (userID)
- search_contains_hashtag
- statuses_update (userID)
- friendships_destroy (userID)
- friendships_create (userID)

# Workload ops

- statuses_update
  - Find the next tweet ID for the user
  - Add the tweet to Tweets entity
  - for each user's follower update its timeline
  - update user's timeline.
- friendships_create and friendships_destroy
  - update the list of followers for the user
  - update the list of people following the new or old followed user
  - recompute the user's timeline.

# Workload generation

- The interleaving of operations take into account previous studies and discussions that took place during Twitter's Chirp conference (the Twitter official developers conference).
- Defines a think-time between operations.
- Next operation is randomly chosen with following probabilities per operation:

# Ops Probabilities

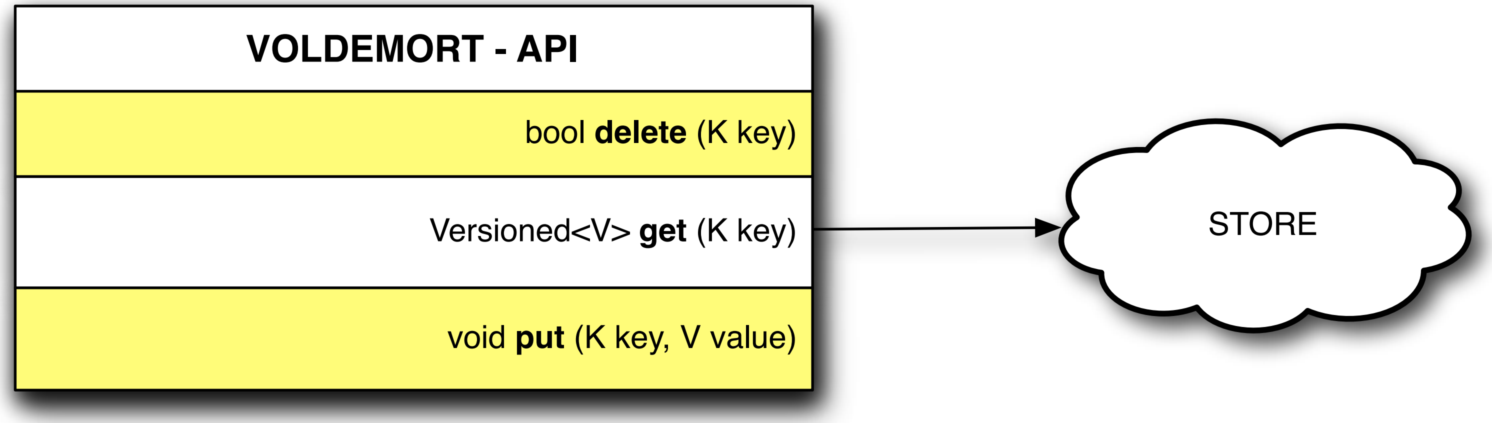
| Operation | Probability |
|---|---|
| search_contains_hashtag | 15% |
| statuses_mentions | 25% |
| statuses_friends_timeline | 50% |
| statuses_update | 5% |
| friendships_create | 2.5% |
| friendships_destroy | 2.5% |

# Existing Implementations

- Workload already implemented for:
  - Voldemort
  - Cassandra
  - Clouder
  - MySQL

# Why Voldemort?

- Voldemort represents a family of row-based stores with a simpler data model and API with only puts and gets (e.g. Amazon's Dynamo).
- Simple mapping of key to value.
- Values are treated as opaque array of bytes.



**VOLDEMORT - API**

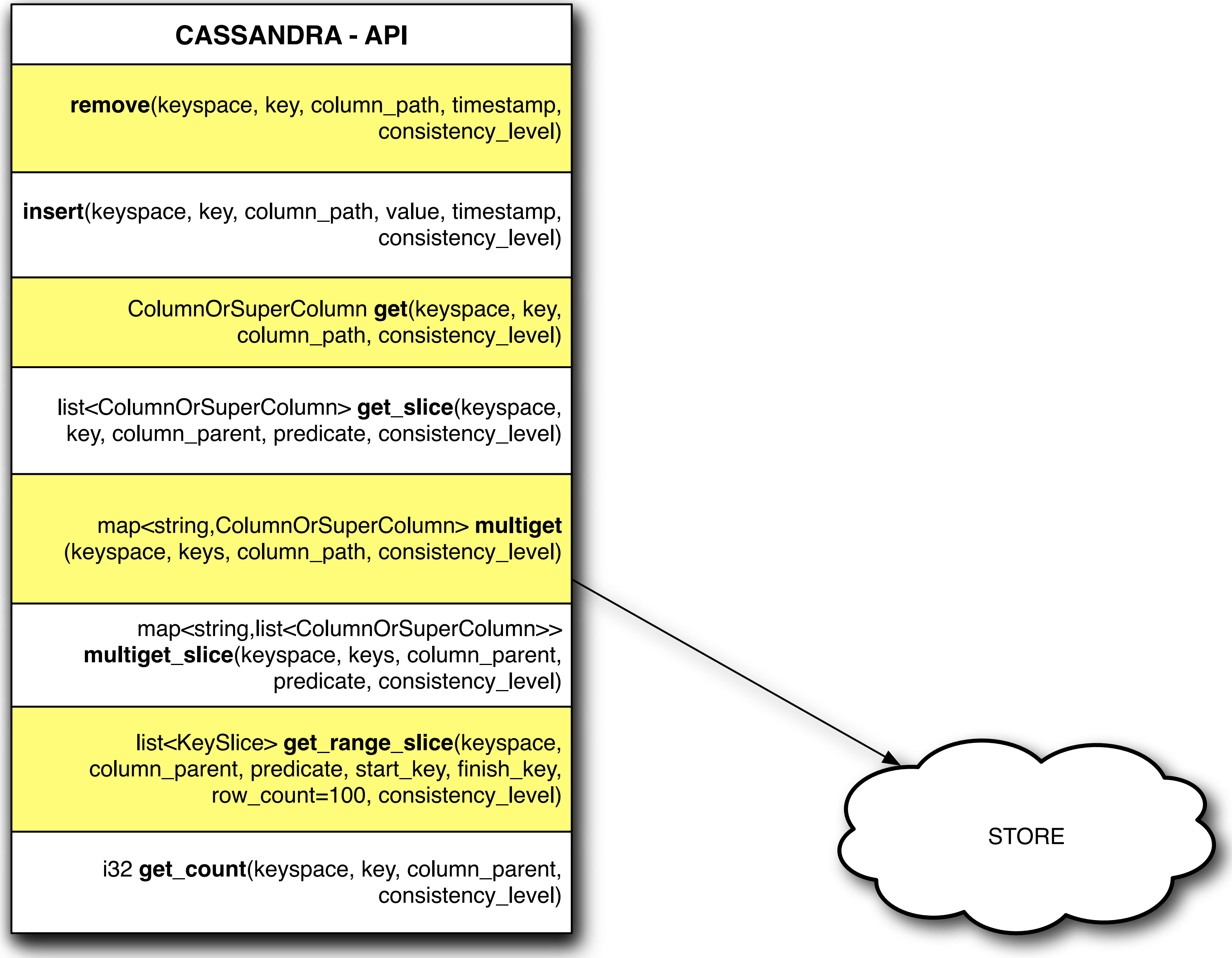
bool **delete** (K key)

Versioned<V> **get** (K key)

void **put** (K key, V value)

STORE

# Why Cassandra?



**CASSANDRA - API**

**remove**(keyspace, key, column_path, timestamp, consistency_level)

**insert**(keyspace, key, column_path, value, timestamp, consistency_level)

ColumnOrSuperColumn **get**(keyspace, key, column_path, consistency_level)

list<ColumnOrSuperColumn> **get_slice**(keyspace, key, column_parent, predicate, consistency_level)

map<string,ColumnOrSuperColumn> **multiget** (keyspace, keys, column_path, consistency_level)

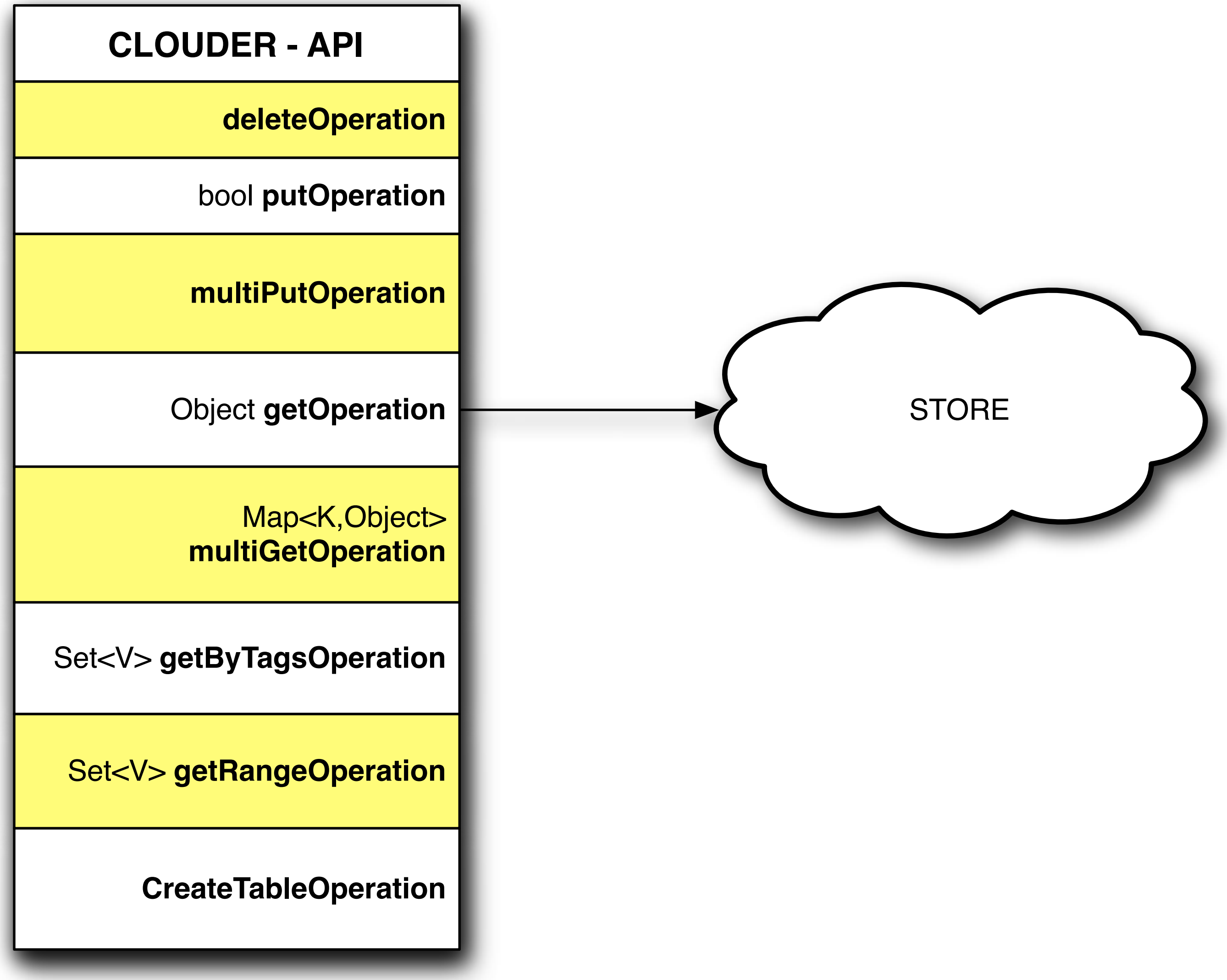
map<string,list<ColumnOrSuperColumn>> **multiget_slice**(keyspace, keys, column_parent, predicate, consistency_level)

list<KeySlice> **get_range_slice**(keyspace, column_parent, predicate, start_key, finish_key, row_count=100, consistency_level)

i32 **get_count**(keyspace, key, column_parent, consistency_level)

- Cassandra offers a different data model: it is column oriented (ColumnFamilies, SuperColumns, etc much like Google's BigTable) .
- Offers a higher level API with range operations.

# Why Clouder?


CLOUDER - API

deleteOperation

bool **putOperation**

**multiPutOperation**

Object **getOperation**

Map<K,Object>
**multiGetOperation**

Set<V> **getByTagsOperation**

Set<V> **getRangeOperation**

**CreateTableOperation**

STORE

- Clouder offers a API with puts, gets as well as search and multi-tuple operations.
- Extends the data model of previous tuple stores with tags, that allows to establish arbitrary relations among tuples.
- Takes advantage of tuple correlation in terms of operations and how partitioning is made.

# Why MySQL?

- Provides a baseline to compare with NoSQL databases.
- At the same time, will assess the suitableness of RDBMS to today's Social Applications.

# Conclusion and Future Work

- A realistic workload for simulating today's high demanding social applications.
- Easy to adapt to the available databases.

- Compare the different databases, in terms of performance, scalability and fault tolerance.
- Add another implementation: VoltDB

Thank you!