# Benchmarking with YCSB in the context of a micro blogging domain

**— Bachelor Thesis —**

Databases and Information Systems Group
Department of Informatics
Faculty of Mathematics, Informatics and Natural Sciences
University of Hamburg

| | |
|---|---|
| Author: | Mirko Köster |
| E-Mail-address: | 0mkoeste@informatik.uni-hamburg.de |
| Mat. No.: | 5616739 |
| Course of Studies: | Bsc. Informatics |
| | |
| 1st assessor: | Prof. Dr. Norbert Ritter |
| 2nd assessor: | Wolfram Wingerath |

Hamburg, January 6th, 2015

# Contents

# List of Figures

# List of Tables

# List of listings

# 1. Introduction

The foundation of *Relational Database Management Systems* (RDBMS) has been introduced by Codd in 1970 [Cod70]. Since then, the performance of available systems based on this relational model has been analyzed and benchmarked (e.g. using the benchmarks from TPC (Transaction Processing Performance Council)).

In 2009 the term NoSQL (meaning *Not only SQL* and should not be interpreted as *No SQL*) was introduced by Johan Oskarsson [Fow15] to describe the then upcoming Database Management Systems (DBMS) not based on the relational model. NoSQL DBMS don't share one common data model. The way how these systems organize and store their data varies from simple *key-value-stores* to *document-oriented databases* to *wide-column datastores*. Key-value-stores simply map one key to one value, where the value itself can often be of a more complex type. Wide-column datastores work column oriented (as opposed to RDBMS, which work row oriented). Each table can consist of millions of (sparse) columns - hence the name wide-column. Document-oriented databases are able to store complex objects (referred to as *documents*), where complex means that documents can be nested (i.e. contain other documents). Additionally most NoSQL DBMS don't offer strong ACID (Atomicity, Consistency, Isolation, Durability) properties like transactions nor join operations like traditional RDBMS do.

As these new datastores are so different from RDBMS and from each other, *apple to apple* comparison of their performance characteristics is not trivial. On the other hand these characteristics are important when one has to choose a DBMS in a given context.

The *Yahoo! Cloud Serving Benchmark* (YCSB) was introduced in 2010 by Cooper et.al. in [CST$^+$10] to address this issue. They assume a simplistic data model which is not suited for most applications and define *workloads* to specify the proportions of CRUD-operations (create, read, update and delete) and other properties neccessary to execute the benchmark. The authors "did not attempt to exactly model a particular application or set of applications", but their "goal was to examine a wide range of workload characteristics" [CST$^+$10, Chapter 4]. Furthermore they state that a "primary goal of YCSB is extensibility" [CST$^+$10, Chapter 5.2].

## 1.1. Motivation and scope of this bachelor thesis

Today YCSB is the established benchmarking suite for NoSQL databases. But due to the rather simplistic data model, the results generated by the benchmark may not be representative for a more realistic (real world) data model. But writing a new benchmark is a very time consuming effort. And as the authors claim that YCSB is extensible, it would be interesting to see if it possible to extend it with arbitrary data models.

The goal of this bachelor thesis is to test how feasible it is to extend YCSB with one new data model, namely in the context of the micro blogging domain. As a proof of concept a prototype will be developed. To reduce the development effort, besides [CST$^+$10] this bachelor thesis is based on [WKF14], which describes how one could implement a Twitter-clone (Twitter[1] is an existing micro blogging service) based on MongoDB[2] as its data store. Part of that project was to develop a working prototype.

Although YCSB is about benchmarking, it is not a goal of this bachelor thesis to perform actual benchmarks and compare several databases exploiting the new data model, but solely to perform a feasability study of the extensibility of YCSB.

## 1.2. Chapter outline

The next chapter is about the current state of research. First [CST$^+$10] is presented in detail. Then related work is discussed.

Chapter 3 introduces the micro blogging domain and its data model, entities and actions. Chapter 4 explains the practical part of this bachelor thesis. It is about extending YCSB. Then in chapter 5 conclusions are drawn and possible future works are presented.

[1]see `https://twitter.com/` for details
[2]see `http://www.mongodb.org/` for details

# 2. State of research

## 2.1. Yahoo! Cloud Serving Benchmark

In 2010 Cooper et.al. presented [CST+10]. They proposed a new benchmark suite called the *Yahoo! Cloud Serving Benchmark* (YCSB) to address those new DBMS, which where published recently at that time and had new/different properties than the established RDBMS. While it was possible to compare these new DBMS qualitatively with each other or existing RDBMS, it was hard to do this comparison quantitatively.

The benchmark defined a simplistic data model (see figure 2.1), which represents a key-value-store. This model uses one entity, which is called *User* and has ten attributes per default. Additionally they used *workloads* to define how the benchmark should be executed. A workload has properties like *the number of fields per record* (fieldcount) or the *proportion of read, write and update operations to perform* (readproportion, insertproportion and updateproportion respectively). For a comprehensive list of properties including their description see appendix A.

The suite comes with support for several DBMS like Google BigTable [CDG+08], Apache HBase[1], Apache Cassandra[2], MongoDb, Redis[3] and others [Wik15].

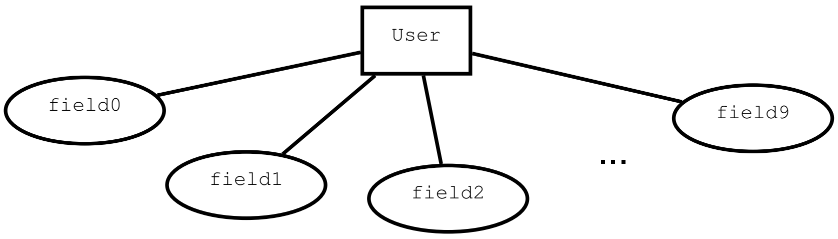

Figure 2.1.: Conceptional data model of YCSB's database

Most NoSQL DBMS make tradeoffs like *optimizing for reads* or *optimizing for writes*, *latency* versus *durability* or *synchronous replication* versus *asynchronous replication*. The workloads they defined for YCSB were designed to "explore these tradeoffs directly" [CST+10, Chapter 4]. Table 2.1 shows these tradeoffs for a small selection of DBMS. BigTable for instance is optimized for fast writes, durability and synchronous replication. Its data model is column oriented.

[1]see `http://hbase.apache.org/` for details
[2]see `http://cassandra.apache.org/` for details
[3]see `http://redis.io/` for details

| System | Read/Write optimized | Latency/durability | Sync/async replication | Row/column |
|---|---|---|---|---|
| PNUTS | Read | Durability | Async | Row |
| BigTable | Write | Durability | Sync | Column |
| HBase | Write | Latency | Async | Column |
| Cassandra | Write | Tunable | Tunable | Column |
| Sharded MySQL | Read | Tunable | Async | Row |

Table 2.1.: YCSB: Design decisions of various systems (recreated from [CST$^+$10, Table 1])

YCSB has five predefined workloads ($A$ to $E$). Table 2.2 shows these workloads and their most important settings. Workload $A$ for instance performs 50% read and 50% update operations. The key chosen to identify a record comes from a zipfian distribution, meaning that some keys are very likely to occur while others are not (YCSB supports the following distributions: uniform, zipfian, latest, multinomial [CST$^+$10, Chapter 4.1]). An application example that matches this workload could be a session store recording recent actions in a user session.

| Workload | Operations | Record selection | Application example |
|---|---|---|---|
| A—Update heavy | Read: 50% Update: 50% | Zipfian | Session store recording recent actions in a user session |
| B—Read heavy | Read: 95% Update: 5% | Zipfian | Photo tagging; add a tag is an update, but most operations are to read tags |
| C—Read only | Read: 100% | Zipfian | User profile cache, where profiles are constructed elsewhere (e.g., Hadoop) |
| D—Read latest | Read: 95% Update: 5% | Latest | User status updates; people want to read the latest statuses |
| E—Short ranges | Read: 95% Update: 5% | Zipfian / Uniform | Threaded conversations, where each scan is for the posts in a given thread (assumed to be clustered by thread id) |

Table 2.2.: YCSB: Workloads in the core package (recreated from [CST$^+$10, Table 2])

Running the benchmark with a given workload against several DBMS, one can compare their *performance* or how well those systems *scale*.

YCSB is released as open source and is considered extensible. Fig 2.2 shows its architecture.

The so called *client* can be run multithreaded. During a run performance criteria are recorded and presented afterwards by the *Stats* module. The *Workload Executor* as well as the *DB Interface Layer* are the components that can easily be extended by third parties.

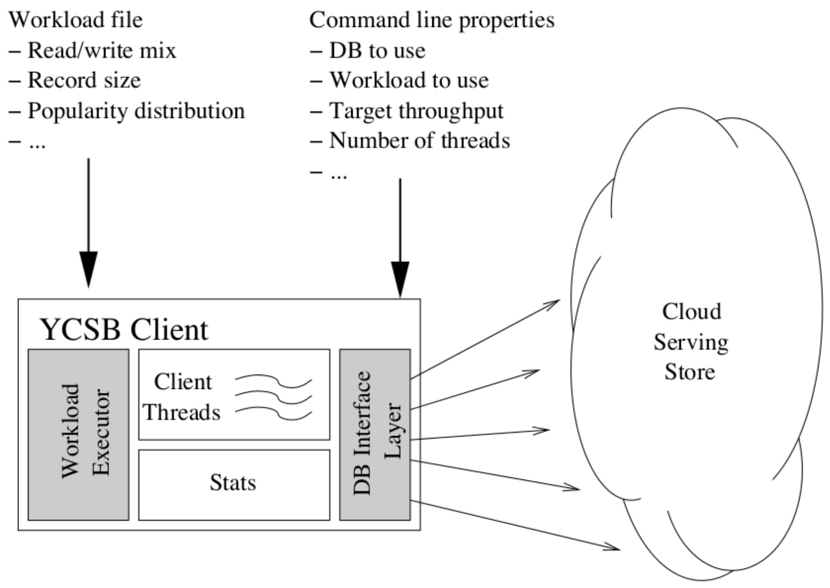

Figure 2.2.: YCSB client architecture [CST+10, Chapter 5, Figure 2]

YCSB is the foundation or inspiration of several other works in this area.

## 2.2. related work

Before YCSB was released Pavlo et.al published [PPR+09] in 2009. In that paper they describe and compare *parallel SQL* and the *MapReduce* (MR) paradigm.

YCSB++ [PPR+11] released by Patil et al. in 2011 is an extension of YCSB. They focus on two HDFS[4]-based, Google BigTable [CDG+08] like table stores (Apache HBase and ACCUMULO[5]). In addition to what YCSB has to offer YCSB++ is looking at the aspects *parallel testing*, *weak consistency*, *table pre-splitting for fast ingest*, *bulk loading using hadoop*, *server-side filtering* and *access control.*

BG [BG13] is a benchmark suite inspired by the beforementioned two papers. Barahmand and Ghandeharizadeh use a more realistic data schema (see figure 2.3), modelling a social network comparable to a simplified Facebook[6]. Furthermore they assume a *service level agreement* (SLA[7]), measure different ratings called *social action rating* (SoAR) and *socialities* (concurrent users) and cover *stale data.*

[4]see `http://en.wikipedia.org/wiki/Apache_Hadoop#HDFS` for details
[5]see `https://accumulo.apache.org/` for details
[6]see `https://www.facebook.com/` for details
[7]see `http://en.wikipedia.org/wiki/Service-level_agreement` for details

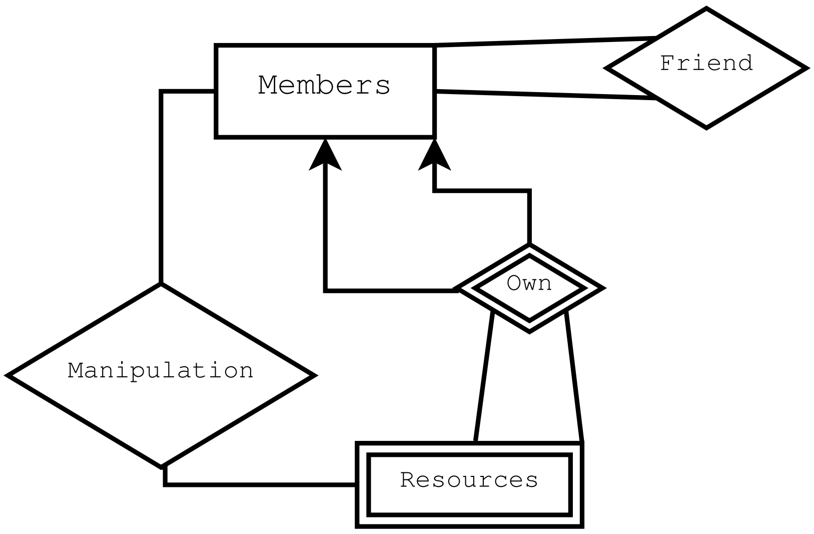

Figure 2.3.: Conceptional data model of BG's database (recreated from [BG13, Chapter 1, Figure 1.a])

A broader overview of recent and ongoing research in this area was published by Friedrich et.al in 2014 [FWGR14]. This overview is organized by *what* is benchmarked. Topics are *latency and throughput*, *availability*, *consistency*, *staleness*, *ordering guarantees*, *transactions* and *consistency in the face of partitions*. Additionally they present their approach inspired by YCSB++ as well as open challenges.

# 3. Data model of the micro blogging domain

This chapter introduces the data model of a micro blogging domain. This model will be presented in two ways. First based on the traditional relational model and then using a document-oriented store. The data model and possible user actions described in this chapter are presented in more detail in [WKF14, Chapter 4].

## 3.1. Data modelling in NoSQL

Most NoSQL DBMS don't offer join operations and/or ACID[1] transactions like traditional RDBMS do. If one needs the special capabilities such a NoSQL data store has to offer (e.g. scaling out), modelling the domain is usually significantly different from how it used to be done in the relational world. Instead of normalizing[2] the schema, data is often stored redundantly. That is due to the lack of join operations and/or to optimize for (read) performance. Another aspect is that some systems don't guarantee strong consistency as in ACID, but one has to deal with eventual consistency.

As the data model has to be highly optimized for performance when a project needs the scaling capabilities of NoSQL data stores, the process of data modelling is to identify the most common and/or expensive queries in the given domain and then create the data model accordingly.

## 3.2. Entities and actions

This section describes the entities and user actions introduced in [WKF14] for the micro blogging domain. There are three entities in the domain. *Users* of the service, *tweets* are the microblogs and *tags* which are used to annotate tweets.

Furthermore there is the concept of a user's *stream of tweets*. This stream contains all tweets of users she follows.

The possible actions users can perform are:

- a user can write a tweet.
    - a user can add one or more tags to a tweet.
    - a user can mention one or more other users in a tweet.

[1]see `http://en.wikipedia.org/wiki/ACID` for details
[2]see `http://en.wikipedia.org/wiki/Database_normalization` for details

- a user $A$ can follow another user $B$.
  this means, that $B$'s tweets are visible in $A$'s stream of tweets
- a user $A$ can unfollow another user $B$ if $A$ was already following $B$.
  this removes $B$'s tweets from $A$'s stream of tweets
- a user can read a user's details.
- a user can read one or more tweets.
- a user can read his stream of tweets.

## 3.3. Entity relationship diagram of the micro blogging domain

Figure 3.1 shows an ER diagram of the domain as described in this chapter.

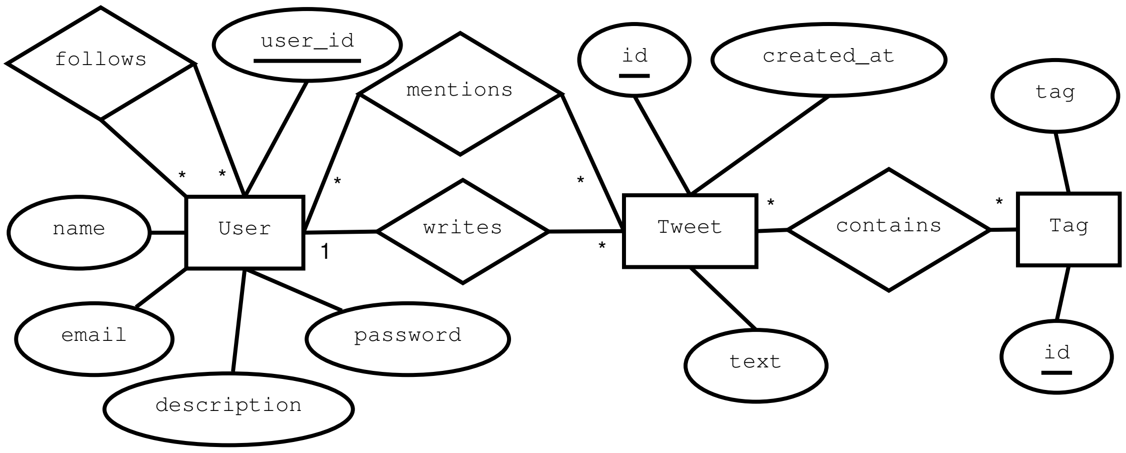

Figure 3.1.: Conceptional data model of the microblogging domain

Explanation of the relations from figure 3.1:

- writes: a tweet is written by *exactly one* user, a user can write *many* tweets.
- contains: a tweet can contain *many* tags, a tag can appear in *many* tweets.
- mentions: a user can mention *many* users, a user can be mentioned in *many* tweets.
- follow: a user can follow *many* users, a user can be followed by *many* users.

## 3.4. Document schema

This section describes how the domain model is represented in the document-oriented data store MongoDB. Listings 3.1 and 3.2 show that users and tweets are stored as simple documents in the corresponding collections.

The user's streams of tweets are stored in the collection *buckets* (see listing 3.3). A bucket bolongs to exactly one user $A$ and stores up to 100 copies of tweets from the users who $A$ is following. Users who follow a lot of other users usually have several buckets to store all those tweet copies. The buckets of one user are organized as a (virtual) double linked list for fast access using the fields *previousBucket* and *nextBucket* as pointers. This list is sorted by the timestamps of the containing tweets. For further details see [WKF14, Chapter 4.2].

This concept is neccessary for fast access, as MongoDB does not offer join operations. That would lead to multiple queries to get that data. This is avoided by storing the tweet copies redundantly. In the relational model, this information is implicitly available via join operations.

```
"user": {
  "_id": "string, unique",
  "name": "string, optional",
  "description": "string, optional",
  "password": "string",
  "email": "string",
  "countTweet": "number",
  "countFollowing": "number",
  "countFollowers": "number"
}
```

Listing 3.1: MongoDB: schema of collection 'users'

### 3.4.1. Asynchronous processing of actions

Most of the actions are performed asynchronously, e.g. if user $A$ follows user $B$, $A$ is marked as follower of $B$. The user can continue using the service immediatly, he does not have to wait for the action to be completed. In the background $B$'s tweets are copied to $A$'s stream of tweets by the *queue processing component* (see [WKF14, Chapter 6]).

```
"tweet": {
  "_id": "ObjectId, unique",
  "author": "string, references a user",
  "authorName": "string, optional",
  "timestamp": "ISODate",
  "text": "string",
  "tags": [
    "string"
    ],
  "users": [
    "string, references a user"
    ]
}
```

Listing 3.2: MongoDB: schema of collection 'tweets'

```
"bucket": {
  "_id": "ObjectId, unique",
  "userId": "string, references a user",
  "upperBound": "ObjectId, references a tweet",
  "lowerBound": "ObjectId, references a tweet",
  "nextBucket": "ObjectId, references a bucket",
  "previousBucket": "ObjectId, references a bucket",
  "tweetCount": "number",
  "tweets": [
    {
      "masterTweet": "ObjectId, references a tweet",
      "timestamp": "ISODate",
      "author": "string, references a user",
      "authorName": "string, optional",
      "text": "string"
    }
    ]
}
```

Listing 3.3: MongoDB: schema of collection 'buckets'

## 3.5. Mapping actions to the two data models

This section compares queries in the relational model (assuming ER diagram 3.1) and queries using the document schema from section 3.4 qualitatively.

For this purpose six of the actions from section 3.2 are presented and analyzed for each model. You can assume that indices are used in both models where appropriate. Furthermore we assume that there are millions of users and billions of tweets to be stored. This is important because not all data could fit into main memory of one server. Thus many queries would require random I/O to disk, increasing response times significantly.

### 3.5.1. Relational algebra

The queries for the relational model are presented using relational algebra[3]. The queries for the first three actions (3.1) to (3.3) are not expensive. But note that query (3.1) delivers just basic information about the user. Including statistics like *number of tweets*, *number of followers* and *number of followees* would require a different, more sophisticated query using several joins, which would make the query more expensive.

The queries for the next three actions (3.4) to (3.6) require joins, which would make the queries more expensive considering the assumptions from before.

- read user 'x':

$$\pi_{user\_id,description,name}(\sigma_{user\_id=x}(User)) \tag{3.1}$$

- read tweet 'x':

$$\pi_{id,created\_at,author,text}(\sigma_{id=x}(Tweet)) \tag{3.2}$$

- read tweets by user 'x':

$$\pi_{id,created\_at,author,text}(\sigma_{user\_id=x}(Tweet)) \tag{3.3}$$

- read tweets with mentioned tag 'x':

$$\pi_{id,created\_at,author,text}(Tweet \bowtie_{id=tweet\_id} Contains \bowtie_{tag\_id=id} (\sigma_{tag=x}(Tag))) \tag{3.4}$$

- read tweets with mentioned user 'x':

$$\pi_{id,created\_at,author,text}(Tweet \bowtie_{id=tweet\_id} (\sigma_{user\_id=x}(Mentions))) \tag{3.5}$$

- read stream from user 'x':

$$\pi_{id,created\_at,author,text}(\sigma_{user\_a=x}(Follows) \bowtie_{user\_b=author} Tweet) \tag{3.6}$$

[3]see `http://en.wikipedia.org/wiki/Relational_algebra` for details

### 3.5.2. MongoDB queries

Please note that covering the query language[4] of MongoDB is beyond the scope of this bachelor thesis. The queries presented in this section should nevertheless be easy to comprehend. *find* and *findOne* both expect two parameters. The first one corresponds to a *selection* ($\sigma$), the second one to a *projection* ($\pi$).

As you can see in listings 3.4 to 3.9, all six queries are simple lookups[5]. Additionally the MongoDB query 3.4 includes the statistics that where missing in query (3.1) from last section.

```
db.users.findOne(
  {_id: "x"},
  {_id: 1, name: 1, description: 1, countTweet: 1,
   countFollowing: 1, countFollowers: 1 }
)
```

Listing 3.4: MongoDB query: read user 'x'

```
db.tweets.findOne(
  {_id: "x"},
  {_id: 1, timestamp: 1, author: 1, text: 1}
)
```

Listing 3.5: MongoDB query: read tweet 'x'

```
db.tweets.find(
  {author: "x"},
  {_id: 1, timestamp: 1, author: 1, text: 1}
)
```

Listing 3.6: MongoDB query: read tweets by user 'x'

[4]see `http://docs.mongodb.org/manual/tutorial/query-documents/` for details

[5]Using MongoDB's transparent sharding functionality (which does not require distributed locking mechanisms) and several machines, data could be kept in main memory completely, thus further improving response times.

```
db.tweets.find(
  {tags: "x"},
  {_id: 1, timestamp: 1, author: 1, text: 1}
)
```

Listing 3.7: MongoDB query: read tweets with mentioned tag 'x'

```
db.tweets.find(
  {users: "x"},
  {_id: 1, timestamp: 1, author: 1, text: 1}
)
```

Listing 3.8: MongoDB query: read tweets with mentioned user 'x'

```
db.buckets.find(
  {userId: "x"},
  {tweets: 1}
)
```

Listing 3.9: MongoDB query: read stream from user 'x'

# 4. Extending YCSB in the context of the micro blogging domain

This chapter explains how one can extend YCSB and shows how this was done for the prototype in the micro blogging domain. Afterwards you find detailed information how to run the prototype.

## 4.1. Extending YCSB in general

“A key design goal of [the] tool is extensibility” [CST$^{+}$10, Chapter 5]. The YCSB benchmarking suite is written in Java and available under the open source license *Apache License, Version 2.0*[1]. Figure 2.2 shows the YCSB client architecture. The two components, which are highlighted in gray, are meant to be easily extended.

The so called “DB Interface Layer” is responsible for translating operations into actual queries for a given DBMS. YCSB comes with implementations for some common DBMS like *cassandra*, *mongodb*, *redis*, *hbase*, as well as for the generalized RDBMS interface *jdbc*[2] (which serves as a generic DB Interface for relational DBMS). Listing 4.1 shows a (shortened) interface of the abstract class “com.yahoo.ycsb.DB”. Lines 7 to 16 show the simple crud operations used by the benchmark. When implementing a new DB class for a DMBS these operations have to be mapped to the semantics of this given DBMS (see [CST$^{+}$10, Chapter 5.2.1]). In their JavaDoc, they say “Rather than dictate the exact semantics of these methods, we recommend you either implement them to match the database’s default semantics, or the semantics of your target application”[3]. Due to the simple data model seen in figure 2.1, the first two parameters of those methods are a table name (by default “usertable”) and a key to identify a record. As we will see this is a big limitation when it comes to extend YCSB with a more sophisticated data model like the micro blogging domain.

The second part from figure 2.2 that is easy to extend, is the “Workload Executor”. One basically has two options how to extend this layer (for details see [CST$^{+}$10, Chapter 5.2]). The first is to define new values for the parameters of the existing workloads (see Appendix A). The second is to write a new implementation of the abstract class “com.yahoo.ycsb.Workload”. Listing 4.4 shows a (shortened) interface of this abstract class.

[1]see `http://www.apache.org/licenses/LICENSE-2.0` for details

[2]see `http://de.wikipedia.org/wiki/Java_Database_Connectivity` for details

[3]see `https://github.com/brianfrankcooper/YCSB/blob/5659fc582c8280e1431ebcfa0891979f806c70ed/core/src/main/java/com/yahoo/ycsb/DB.java#L38-L41` for details

## 4.2. Extending YCSB with the micro blogging domain model

Due to the simple data model of YCSB and the rather simple CRUD operations of the benchmark suite, there is no 'natural' way to extend YCSB to use the model discussed in section 3.

The best solution would be to not use the existing classes *DB* and *Workload*, but write something completely new that is better suited to be adjusted to a more complex application setting. But that would mean that the other components of YCSB like the stats module can not be reused either. So this approach would lead to a complete rewrite of the benchmark suite. But this is outside the scope of this bachelor thesis.

As the implementation should just be a proof of concept, another approach is to use the existing classes *DB* and *Workload* with different semantics:

The signature of the methods are unchanged. But as the parameter names are not part of the signature in java ("Two of the components of a method declaration comprise the *method signature* – the method's *name* and the *parameter types*." [Ora15]), they could be renamed. This is not considered good practice, but serves the purpose of a proof of concept.

The details are explained in the next two sections.

### 4.2.1. YCSB - DB for MongoDB and the micro blogging domain

The *DB class* corresponds to the *DB Interface Layer* seen in section 4.1. Instances of this class are threadlocal, meaning each thread has its own instance.

The idea is to first leave the existing code as it is, so no breaking changes with existing implementations for the various DBMS are introduced. And second to introduce a new Java Interface, which offers the same methods as the DB class and reuses the first two parameters of the CRUD operations with different semantics by renaming them and adjusting the JavaDoc comments accordingly. Listing 4.1 shows the original methods from YCSB and listing 4.2 shows the CRUD operations with the new semantics (see Appendix B for a full version including JavaDoc comments). This concept is discussed for the *read* method but applies to the other four methods (*scan*, *update*, *insert* and *delete*) analogously.

#### Redefining semantics for the read operation

The first parameter *String table* is renamed to *String readAction*. Possible values an implementation should be able to handle are *user* and *tweet* for reading a user's details or a (random) tweet of a user. The second parameter *String key* is renamed to *String user* and should contain the user's ID for whom the read operation should be done. Listing 4.3 shows an example call of the read method with the new semantics. This would read the details of user *u123*.

```java
package com.yahoo.ycsb;
public abstract class DB {
  public void setProperties(Properties p);
  public Properties getProperties();
  public void init() throws DBException;
  public void cleanup() throws DBException;
  public abstract int read(String table, String key,
    Set<String> fields, HashMap<String,ByteIterator> result);
  public abstract int scan(String table, String startkey,
    int recordcount, Set<String> fields,
    Vector<HashMap<String,ByteIterator>> result);
  public abstract int update(String table, String key,
    HashMap<String,ByteIterator> values);
  public abstract int insert(String table, String key,
    HashMap<String,ByteIterator> values);
  public abstract int delete(String table, String key);
}
```

Listing 4.1: YCSB: abstract class DB (shortened), Licensed under the Apache License 2.0

```java
package de.mirkokoester.ycsb;
public interface MicrobloggingDB {
  public int read(String readAction, String user, Set<String> fields,
      HashMap<String, ByteIterator> result);
  public int scan(String scanAction, String user, int recordcount,
      Set<String> fields, Vector<HashMap<String, ByteIterator>> result);
  public int update(String updateAction, String user,
      HashMap<String, ByteIterator> values);
  public int insert(String insertAction, String user,
      HashMap<String, ByteIterator> values);
  public int delete(String deleteAction, String user);
}
```

Listing 4.2: YCSB: MicrobloggingDB interface

```java
db.read("user", "u123", fields, result);
```

Listing 4.3: YCSB: calling the read method with new semantics

### 4.2.2. YCSB - Workload for micro blogging domain

The *Workload class* corresponds to the *Workload Executor* seen in section 4.1. The single instance of this class is shared among all threads.

The new implementation of the *Workload class* takes the new semantics of the *MicrobloggingDB interface* into account. *Workload* defines two methods that operate on instances of *DB*:

**doInsert** load data store prior to benchmarking; create defined state

**doTransaction** run benchmark, choose CRUD operation according to properties of workload

```java
package com.yahoo.ycsb;
public abstract class Workload {
  public void init(Properties p) throws WorkloadException;
  public Object initThread(Properties p, int mythreadid,
    int threadcount) throws WorkloadException;
  public void cleanup() throws WorkloadException;
  public boolean doInsert(DB db, Object threadstate);
  public boolean doTransaction(DB db, Object threadstate);
  public void requestStop();
  public boolean isStopRequested();
}
```

Listing 4.4: YCSB: abstract class Workload (shortened), Licensed under the Apache License 2.0

### 4.2.3. Limitations of this approach

This approach has other issues besides its discussed design:

- *DB* implementations with the new semantics are not compatible with existing *Workloads*. And *Workload* implementations with the new semantics are not compatible with existing *DB*s.
- Unfortunately it is not possible to check the correct usage at compile time. Which classes should be used are determined from parameters at run time. The Instances are then created via reflection[4].
- The actual instances of the specified *DB class* are wrapped in a class called *com.yahoo.ycsb.DBWrapper*. This means that one can not even at run time check whether *DB* implements the new interface from listing 4.2.

[4]see `http://en.wikipedia.org/wiki/Reflection_(computer_programming)` for details

Although the prototype, which uses this approach, is fully functional, it is not recommended to extend YCSB using this approach.

## 4.3. Running the extended benchmark

This section shows how to run the prototype. If you don't want to build the project yourself, you can skip the next section and continue with section 4.3.2.

### 4.3.1. Building

YCSB uses Apache Maven[5] v3.x as its build tool. You can invoke the build process on the command line with the following command:

```
mvn clean package
```

### 4.3.2. Running

#### MongoDB

You need a running instance of MongoDB 2.4.x or 2.6.x (either a single node, or a ReplSet or sharded). The easiest way to get started is with a single node. You can install it for your operating system[6] or use Docker[7] to run it in a container[8].

**MongoDB via Docker** Once Docker is up and running, you can start a MongoDB instance as follows:

```
sudo docker run -d -p 27017:27017 -p 28017:28017 --name mongodb \
  dockerfile/mongodb mongod --rest --httpinterface
```

This will create a new container called 'mongodb' using the official MongoDB image and make its ports 27017 and 28017 (webinterface) available on your system. If you need to connect to the database using the MongoDB shell[9] (or command line interface (CLI)), you can do this also with Docker: This command creates a temporary container (–rm) which is linked to your first container 'mongodb' and runs the interactive MongoDB shell and connects to the database using the db 'ycsb'.

[5]see `http://maven.apache.org/` for details
[6]see `http://www.mongodb.org/downloads` for details
[7]see `https://www.docker.com/` for details
[8]see `http://en.wikipedia.org/wiki/Software_container` for details
[9]see `http://docs.mongodb.org/manual/tutorial/getting-started-with-the-mongo-shell/` for details

```
sudo docker run -it --rm --link mongodb:mongodb dockerfile/mongodb \
  bash -c 'mongo mongodb/ycsb'
```

**Tuning MongoDB (indices)** Before running the benchmark, you should create the indices for each collection. The following commands have to be executed within the MongoDB shell:

```
use ycsb
db.users.drop()
db.users.ensureIndex( { _id: 1 } )

db.tweets.drop()
db.tweets.ensureIndex( { _id: 1 } )
db.tweets.ensureIndex( { author: 1, _id: 1 } )

db.usersFollowing.drop()
db.usersFollowing.ensureIndex( { _id: 1, following: 1 } )

db.usersFollower.drop()
db.usersFollower.ensureIndex( { _id: 1, followed: 1 } )

db.events.drop()
db.createCollection("events", {capped: true, size: 200000, max: 10000} )
db.events.insert( {userId: "u0"} )

db.queues.drop()
db.queues.ensureIndex( { _id: 1 } )
db.queues.insert(  { _id : "u0", queue : [] } )

db.buckets.drop()
db.buckets.ensureIndex( { userId: 1, lowerBound: 1 } )
```

Listing 4.5: MongoDB: creating indices

### extended YCSB

This section describes how to run the actual extended benchmark in the context of a micro blogging domain.

First, you have to make sure that the queue processing component (see 3.4.1) is running on the same server as MongoDB and can access the primary on *localhost* and port *27017*:

```
bin/mongoqueueakka
```

**Load phase** Next, you can start loading the database:

```
bin/ycsb load mongodb_microblogging -s \
  -P workloads/workloada_microblogging \
  -p mongodb.database=ycsb -p mongodb.writeConcern=normal \
  -p mongodb.maxconnections=100 -p mongodb.url=mongodb://localhost:27017
```

This may take some while. But even if the suite finished, it may take the queue processing component a while longer to process the actions asynchronously. One can watch the CPU consumption of that process to tell when it is done.

**Run phase** Now, the benchmarking can be started with:

```
bin/ycsb run mongodb_microblogging -s \
  -P workloads/workloada_microblogging \
  -p mongodb.database=ycsb -p mongodb.writeConcern=normal \
  -p mongodb.maxconnections=100 -p mongodb.url=mongodb://localhost:27017
```

When it is done, you get the same statistics as in the original YCSB benchmark suite. The output looks like listing 4.6.

```
...
scan: action posts, user u751, recordcount 21, fields: null
scan: action posts, user u111, recordcount 22, fields: null
2015-01-05 15:52:37:594 1 sec: 1350 operations; 972.62 current ops/sec;
[READ AverageLatency(us)=488.21]
[DELETE AverageLatency(us)=695.34]
[CLEANUP AverageLatency(us)=1207]
[INSERT AverageLatency(us)=973]
[UPDATE AverageLatency(us)=1423.31]
[SCAN AverageLatency(us)=1602.96]
[OVERALL], RunTime(ms), 1421.0
[OVERALL], Throughput(ops/sec), 950.0351864883885
[READ], Operations, 393
[READ], AverageLatency(us), 488.2086513994911
[READ], MinLatency(us), 102
[READ], MaxLatency(us), 37251
[READ], 95thPercentileLatency(ms), 0
[READ], 99thPercentileLatency(ms), 3
...
```

Listing 4.6: YCSB: sample output of the extended benchmark

**optional: running the website from [WKF14]** The website from [WKF14] can be run on the same machine as MongoDB with the following command:

```
bin/website
```

Now one can access the website in the browser at *http://<server>:9000*; e.g. `http://localhost:9000`

**properties** Similar to the original YCSB benchmarking suite (see appendix A), you can tune the extended version with some properties:

- general

  **passive_equals_active** should there be 2 groups[10] of users with different behaviour? if true, the next 4 properties are being ignored (defaults to false)

  **activePassiveRatioFollowing** proportion of active and passive users performing action *follow* (defaults to 0.4)

[10] *active users* tend to write new content, *passive users* consume more content than they produce

**activePassiveRatioFollowed** proportion of active and passive users being *followed* (defaults to 0.75)

**activePassiveRatioTweeting** proportion of active and passive users performing action *tweet* (defaults to 0.8)

**activePassiveRatioBeingMentioned** proportion of active and passive users being *mentioned* (defaults to 0.8)

**workload** the Java Workload class to be used (defaults to *de.mirkokoester.ycsb.microblogging.workloads.CoreWorkload*

- load phase

  **usercount_generate** how many new *user accounts* should be created?

  **tweetcount_generate** how many new *tweets* should be created?

  **followcount_generate** how many *follow* actions should be performed?

- run phase

  **usercount_insert** how many new *user accounts* should be created?

  **tweetcount_insert** how many new *tweets* should be created?

  **usercount_read** how many *read user* actions should be performed?

  **tweetcount_read** how many *read tweet* actions should be performed?

  **tweetcount_delete** how many *delete tweet* actions should be performed?

  **followcount_update** how many *follow* actions should be performed?

  **unfollowcount_update** how many *unfollow* actions should be performed?

  **usercount_scan** how many *read tweets* actions should be performed?

  **tweetcount_scan** how many *read tweet stream* actions should be performed?

# 5. Conclusion

The whole codebase of YCSB focusses on the simple data model of a key-value-store and simple CRUD operations.

It is relatively easy to extend YCSB with new Interfaces to new databases using this data model. It is even easier to adjust the existing workloads to one's needs. And one can create eevn complete new workloads writing Java code. But again, just when using the simple data model.

If the context of your application you want to benchmark several DBMS for also matches those assumptions, you can even adopt YCSB to that context. If that is not the case, the restrictions discussed in the last chapter keep you from extending YCSB to your context. Additionally changing the existing classes so that the benchmark suite matches your context is not a good idea either as this would introduce breaking changes with the existing implementations of the class DB.

Although the implementation discussed in the last chapter works and uses a data model of a micro blogging domain, it should be seen as aproof of concept with a design that is not recommended for production use.

This leads to the conclusion, that YCSB is not easily extendable with other data models, as it focusses to strongly on the data model of a key-value-store.

## 5.1. Outlook

It would be interesting to design a new benchmarking suite inspired by YCSB, but takes the conclusions from this work into account and aims for extensibility even with different data models. The best case would be to be able to use arbitrary data models. If that is not suitable, at least a big enough subset of data models of most existing DBMS should be supported.

# Appendices

# A. original YCSB

The creators of the YCSB benchmark suite have created a wiki[1] where one can find most information needed to run[2] the benchmark.

The core workload package, that comes with YCSB, can be configured with some properties[3]:

**fieldcount** the number of fields in a record (default: 10)

**fieldlength** the size of each field (default: 100)

**readallfields** should reads read all fields (true) or just one (false) (default: true)

**readproportion** what proportion of operations should be reads (default: 0.95)

**updateproportion** what proportion of operations should be updates (default: 0.05)

**insertproportion** what proportion of operations should be inserts (default: 0)

**scanproportion** what proportion of operations should be scans (default: 0)

**readmodifywriteproportion** what proportion of operations should be read a record, modify it, write it back (default: 0)

**requestdistribution** what distribution should be used to select the records to operate on – uniform, zipfian or latest (default: uniform)

**maxscanlength** for scans, what is the maximum number of records to scan (default: 1000)

**scanlengthdistribution** for scans, what distribution should be used to choose the number of records to scan, for each scan, between 1 and maxscanlength (default: uniform)

**insertorder** should records be inserted in order by key ("ordered"), or in hashed order ("hashed") (default: hashed)

**operationcount** Number of operations to perform.

**maxexecutiontime** Maximum execution time in seconds. The benchmark runs until either the operation count has exhausted or the maximum specified time has elapsed, whichever is earlier.

**table** The name of the table (default: usertable)

[1]see `https://github.com/brianfrankcooper/YCSB/wiki` for details
[2]see `https://github.com/brianfrankcooper/YCSB/wiki/Running-a-Workload` for details
[3]see `https://github.com/brianfrankcooper/YCSB/wiki/Core-Properties` for details

# B. YCSB extended

```java
package de.mirkokoester.ycsb;

import java.util.HashMap;
import java.util.Set;
import java.util.Vector;
import com.yahoo.ycsb.ByteIterator;

/**
 * The MicrobloggingDB interface has no additional methods or fields
 * and serves only to identify the semantics of being
 * a DB [com.yahoo.ycsb.DB] in the context of the
 * micro blogging domain.
 *
 * @note mix in with com.yahoo.ycsb.DB
 */
public interface MicrobloggingDB {

  /**
   * Read from the database. Each field/value pair from the result(s)
   * will be stored in a HashMap.
   * [com.yahoo.ycsb.DB#read(java.lang.String, java.lang.String,
   *   java.util.Set, java.util.HashMap)]
   * @param readAction The read action to perform.
   *   can be one of "user", "tweet"
   * @param user The user for which the operation should be executed.
   * @param fields The list of fields to read,
   *   or null for all of them
   * @param result A HashMap of field/value pairs for the result
   * @return Zero on success, a non-zero error code on error
   *   or "not found".
   */
  public int read(String readAction, String user, Set<String> fields,
      HashMap<String, ByteIterator> result);
```

Listing B.1: YCSB: interface for DB with new semantics (Part 1)

```
/**
 * Scan the Database.
 * [com.yahoo.ycsb.DB#scan(java.lang.String, java.lang.String, int,
 *   java.util.Set, java.util.Vector)]
 *
 * @param scanAction The scan action to perform.
 *   can be one of "posts", "stream"
 * @param user The user for which the operation should be executed.
 * @param fields The list of fields to read, or null for all of them
 */
public int scan(String scanAction, String user, int recordcount,
    Set<String> fields, Vector<HashMap<String, ByteIterator>> result);

/**
 * Update an entry in the DB.
 * [com.yahoo.ycsb.DB#update(java.lang.String,
 *   java.lang.String, java.util.HashMap)]
 *
 * @param updateAction The update action to perform.
 *   can be one of "follow", "unfollow"
 * @param user The user for which the operation should be executed.
 */
public int update(String updateAction, String user,
    HashMap<String, ByteIterator> values);

/**
 * creates a new user or inserts a new tweet for the given user.
 * [com.yahoo.ycsb.DB#insert(java.lang.String,
 *   java.lang.String, java.util.HashMap)]
 *
 * @param insertAction The insert action to perform.
 *   can be one of "user", "tweet"
 * @param user The user for which the operation should be executed.
 * @param values keys if insertAction is
 *         'user' - name, description, password, email
 *         'tweet' - text
 */
public int insert(String insertAction, String user,
    HashMap<String, ByteIterator> values);
```

Listing B.2: YCSB: interface for DB with new semantics (Part 2)

```
  /**
   * deletes a tweet of the given user.
   * [com.yahoo.ycsb.DB#delete(java.lang.String, java.lang.String)]
   *
   * @param deleteAction is ignored,
   *   since only tweets are being deleted
   * @param  user The user for which the operation should be executed.
   */
  public int delete(String deleteAction, String user);
}
```

Listing B.3: YCSB: interface for DB with new semantics (Part 3)

# Bibliography

[BG13] Barahmand, Sumita ; Ghandeharizadeh, Shahram: BG: A Benchmark to Evaluate Interactive Social Networking Actions. In: *CIDR*, www.cidrdb.org, 2013

[CDG+08] Chang, Fay ; Dean, Jeffrey ; Ghemawat, Sanjay ; Hsieh, Wilson C. ; Wallach, Deborah A. ; Burrows, Mike ; Chandra, Tushar ; Fikes, Andrew ; Gruber, Robert E.: Bigtable: A Distributed Storage System for Structured Data. In: *ACM Trans. Comput. Syst.* 26 (2008), Juni, Nr. 2, 4:1–4:26. `http://dx.doi.org/10.1145/1365815.1365816`. – DOI 10.1145/1365815.1365816. – ISSN 0734–2071

[Cod70] Codd, E. F.: A Relational Model of Data for Large Shared Data Banks. In: *Commun. ACM* 13 (1970), Juni, Nr. 6, 377–387. `http://dx.doi.org/10.1145/362384.362685`. – DOI 10.1145/362384.362685. – ISSN 0001–0782

[CST+10] Cooper, Brian F. ; Silberstein, Adam ; Tam, Erwin ; Ramakrishnan, Raghu ; Sears, Russell: Benchmarking Cloud Serving Systems with YCSB. In: *Proceedings of the 1st ACM Symposium on Cloud Computing*. New York, NY, USA : ACM, 2010 (SoCC '10). – ISBN 978–1–4503–0036–0, 143–154

[Fow15] Fowler, Martin: *NosqlDefinition*. `http://martinfowler.com/bliki/NosqlDefinition.html`. Version: Januar 2015

[FWGR14] Friedrich, Steffen ; Wingerath, Wolfram ; Gessert, Felix ; Ritter, Norbert: NoSQL OLTP Benchmarking: A Survey. (2014)

[GL02] Gilbert, Seth ; Lynch, Nancy: Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. In: *SIGACT News* 33 (2002), Juni, Nr. 2, 51–59. `http://dx.doi.org/10.1145/564585.564601`. – DOI 10.1145/564585.564601. – ISSN 0163–5700

[Ora15] Oracle: *The Java™ Tutorials - method signatures*. `http://docs.oracle.com/javase/tutorial/java/javaOO/methods.html`. Version: Januar 2015

[PPR+09] Pavlo, Andrew ; Paulson, Erik ; Rasin, Alexander ; Abadi, Daniel J. ; DeWitt, David J. ; Madden, Samuel ; Stonebraker, Michael: A comparison of approaches to large-scale data analysis. In: *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*. New York, NY, USA : ACM, 2009. – ISBN 978–1–60558–551–2, 165–178

[PPR+11] Patil, Swapnil ; Polte, Milo ; Ren, Kai ; Tantisiriroj, Wittawat ; Xiao, Lin ; López, Julio ; Gibson, Garth ; Fuchs, Adam ; Rinaldi, Billie: YCSB++: Benchmarking and Performance Debugging Advanced Features in Scalable Table Stores. In: *Proceedings of the 2Nd ACM Symposium on Cloud Computing*. New York, NY, USA : ACM, 2011 (SOCC '11). – ISBN 978–1–4503–0976–9, 9:1–9:14

[Wik15] Wiki, YCSB G.: *YCSB DBMS Overview*. `https://github.com/briankfrankcooper/YCSB/wiki#overview`. Version: Januar 2015

[WKF14] Witt, Erik ; Köster, Mirko ; Finnern, Malte: Erstellung eines Twitter-Clones mit MongoDB und Elasticsearch / Fachbereich Informatik, Fakultät für Mathematik, Informatik und Naturwissenschaften, Universität Hamburg. Hamburg, Germany, September 2014. – Projektbericht 'Masterprojekt NoSQL'. – 47 S.

## Erklärung

Ich versichere, dass ich die Arbeit selbstständig verfasst und keine anderen, als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internetquellen – benutzt habe, die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Ich bin mit der Einstellung der Bachelor-Arbeit in den Bestand der Bibliothek des Fachbereichs Informatik einverstanden.

Hamburg, den 06.01.2015 ______________________________

Mirko Köster