Column Stores and HBase

Rui LIU, Maksim Hrytsenia

December 2017

Contents

1 Hadoop **2**
1.1 Creation 2

2 HBase **3**
2.1 Column Store Database 3
2.2 HBase Architecture 5
2.3 Basic Commands for HBase 5
2.4 Features 6

3 Project Introduction **7**
3.1 OS choice 7
3.2 DataSet 7
3.3 MySQL and Sqoop 9

4 Application **11**
4.1 Project Set Up 11
4.2 Basic projects and commands using Java API 12
4.3 Data Loading 13
4.4 HBase benchmark comparison 16

5 Conclusion **17**

Chapter 1

Hadoop

1.1 Creation

Nowadays many companies and people from IT are interested in Hadoop because it allows to process big amounts of data. The Hadoop was introduced in 2006, however, the idea of the Hadoop itself was presented to the public in 2003 by company Google [6]. Following that, additional papers that reveal more concrete topics were published [3] and [2]. Almost all of the solutions provided in the paper were directed on handling processing of huge amount of data. Combined this techniques provide the following advantages:

1. Scalability
2. High computing power
3. Fault tolerance
4. Storage and processing speed
5. Lowcost
6. Flexibility

Chapter 2

HBase

HBase is the Hadoop database, a distributed, scalable, big data store. It provides random, realtime read/write access to the big table. HBase is modeled after Google's Bigtable: A Distributed Storage System for Structured Data. It is built on the top of HDFS and aims at hosting very large tables.

2.1 Column Store Database

For traditional row-oriented database, the data will be store row by row. Figure 2.1 shows the physical layout for row-oriented database. Data is stored by rows in the database system.

On the contrary to row-oriented database, column-oriented database divides the tables by columns. Figure 2.2 shows the physical layout of the row-oriented database. Usually, in the database, many of the data are irrelevant to the query. Column store database can solve this problem. Each time when we need to find out a certain value, we just need to go through a certain column, it could avoid the waste of searching for useless columns.

Column store database has four basic elements: *Column Family*, *Column*, *Row Key*, *Timestamp*.

1. **Column Family:** Column family is a collection of columns. Physically,

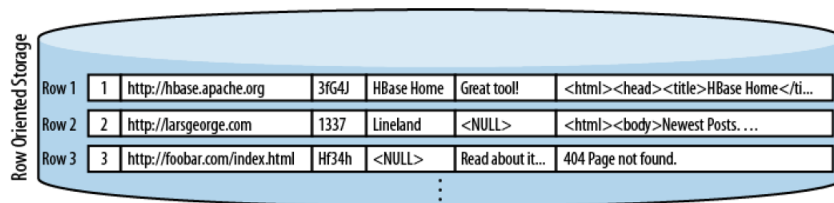

Figure 2.1: Row-oriented database[5]

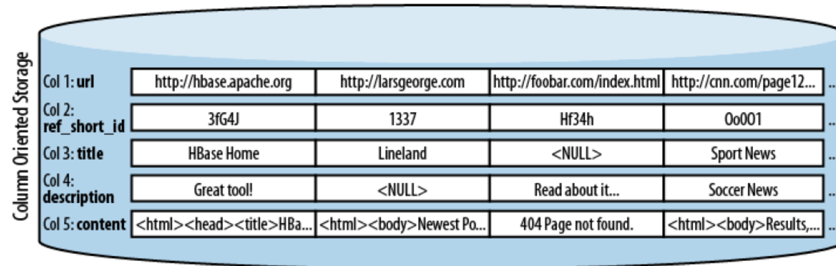

Figure 2.2: Column-oriented database[5]

columns which belong to one family will store together on the file system. So if the columns have the similar access pattern, it's advised to have these columns to be in the same column family. HBase suggests to have no more than three column families in the database sever. What' more, HBase suggests to have same cardinality of the rows in different column families. HBase will split the data rows and store them into different region servers. For instance column family A only has a few rows, but the second column family B has huge amount of the rows. The data will still be split based on number of rows of column family B. It will lead to inefficiency of scanning the the column family A.

2. **Column:** The data in the column-oriented database is divided by columns and store on the file system. Columns in HBase do not have specific data type. Also there is no constraint between each column. This feature provides the efficiency and flexibility of writing data and adding columns into HBase. Unlike row-oriented database, when a query is going to look for a certain value, the database system will just direct to the columns and go through the relevant columns. There will not have join operations in column store database. This feature makes reading faster in big tables.

3. **Row Key:** Row key is a column of the data which is used to connect several column families and columns. With row key and timestamp together, the database system can locate the certain row which is relevent to the query.

4. **Time Stamp:** Every time when a new value is inserted into the table, the value will get a time stamp automatically. Timestamp shows the version of the one cell. From HBase version 0.96, the default number of version is set to 1. If we are going to use different versions of the data, we need to set the max number of the versions manually for each column family.

2.2 HBase Architecture

Figure 2.3 shows the architecture of HBase. It has master server and several region servers. Usually we can use provides Java API to access the data. HBase is built on the top of the HDFS and take the help from ZooKeeper to manage master server and region servers.

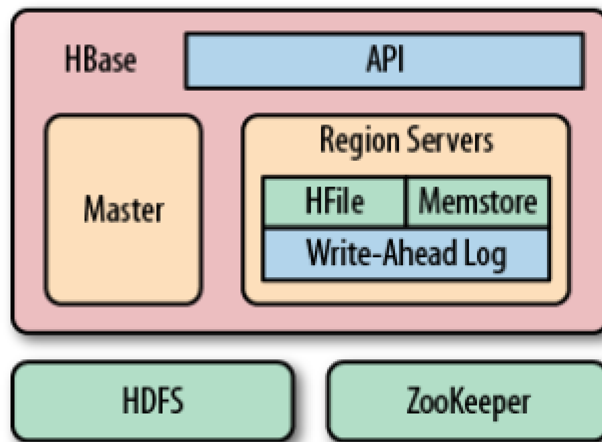

Figure 2.3: HBase Architecture[5]

1. **Master Server:** The master server from HBase is responsible for managing metadata, for instance taking care of the creation of tables, columns and column families. Master server is also in charge of loading balanced regions across region servers. It will unload the busy servers and move regions to less-occupied servers.

2. **Region Server:** The data stored in HBase will be split to different regions. Regions are nothing but sub-tables which are divided based on the user defined column. Clients contact region servers directly for read and write operations.

3. **HDFS:** HBase is built on the top of HDFS. HDFS is a distributed, highly tolerant file system. HBase does not need to do operations to make reliability of the data. It will all be handled by HDFS.

4. **ZooKeeper:** ZooKeeper is a coordination service from Hadoop. It will help master server to assign regions to region servers.

2.3 Basic Commands for HBase

This chapter will introduce how to use HBase shell to do basic access to data.

1. **Start HBase**

   $ cd /usr/local/HBase
   $ bin/start-hbase.sh

2. **Start the master server**

   $ ./bin/local-master-backup.sh start 2

   '2' indicates the number of servers we would like to start.

3. **Start the region server**

   $ ./bin/./local-regionservers.sh start 3

   '3' indicates the number of region servers we would like to start.

4. **Create table**

   $ create 'table','column_family_1','column_family_2'

   In the creation operation, we need to define the name and column family of the table. Columns' names are not needed to define here.

5. **Read data**

   $ get 'table', 'row_key', {COLUMN = 'column_family: column'}

   Get will return the value of columns from certain row.

   $ scan 'table', {COLUMN = 'column_family': column}

   Scan will return the value of column from a range of the rows.

6. **Set versions**

   $ alter 'table', NAME = 'column_family', VERSIONS = 3

   Default number of version is set to 0 at HBase, so we need to manually change the number of versions.

2.4 Features

HBase is modeled after the Google's Bigtable design. It is scalable of storing big table. With the help of MapReduce, HBase can aggregate the data and return a small portion of results to client. It can reduce data transferred over the network. HBase is also highly fault tolerant. HDFS creates replicas and clients do not need to be suffered from the failure of nodes. HBase internally uses harsh table to store the data on file system, so it can provide random access to data.

Chapter 3

Project Introduction

3.1 OS choice

After completing the project requirements we started to search for an appropriate tool how to start to work with the database as quick as possible. First of all, we did want to consider an option of Ubuntu or another Linux distributive without installed Hadoop dependencies from box. The reason is that installation of Hadoop requires a huge number of libraries and it would take a long time to set everything up, so we chose Cloudera CDH. It is an open-source solution that allows to use many Apache licenced products and instruments such as Hadoop Core, Sqoop, Hive, Spark and etc.

3.2 DataSet

After analysing the information about HBase we understood that we need a huge dataset to measure our performance due to the fact that HBase is extremely good in processing big tables. Looking at Facebook example we understood that the dataset with many users and some information related to them would be a perfect finding for us. Luckily, we found a MovieLens dataset [1] that contains more than 20 millions of user *ratings* that was enough to show the best features of HBase. Besides *ratings* we chose to use files *tags* and *movies* as well.At the same time, some data preprocessing was required. For example, movie structure looked at the beginning as on the Figure 3.1. It is easy to see that genres does not look like appropriately for selection. Knowing the fact that HBase supports many versions we decided to get all the genres from that column.

<table>
<tr><th>movieId</th><th>title</th><th>genres</th></tr>
<tr><td>1</td><td>Toy Story (1995)</td><td>Adventure|Animation|Children|Come</td></tr>
<tr><td>2</td><td>Jumanji (1995)</td><td>Adventure|Children|Fantasy</td></tr>
<tr><td>3</td><td>Grumpier Old Men (1995)</td><td>Comedy|Romance</td></tr>
</table>

Figure 3.1: Movie example

Finally, we created four tables in MySql at first. The schema of the tables are as belowing:

1. **Movies**

```
mysql> SHOW COLUMNS FROM movies;
+---------+-------------+------+-----+---------+-------+
| Field   | Type        | Null | Key | Default | Extra |
+---------+-------------+------+-----+---------+-------+
| movieid | int(11)     | NO   | PRI | 0       |       |
| name    | varchar(20) | YES  |     | NULL    |       |
| year    | int(11)     | YES  |     | NULL    |       |
+---------+-------------+------+-----+---------+-------+
```

Figure 3.2: Movies

2. **Genres**

```
mysql> SHOW COLUMNS FROM genres;
+---------+-------------+------+-----+---------+-------+
| Field   | Type        | Null | Key | Default | Extra |
+---------+-------------+------+-----+---------+-------+
| id      | int(11)     | NO   | PRI | 0       |       |
| movieid | int(11)     | YES  |     | NULL    |       |
| genreid | varchar(20) | YES  |     | NULL    |       |
+---------+-------------+------+-----+---------+-------+
```

Figure 3.3: Genres

3. **Ratings**

```
mysql> SHOW COLUMNS FROM ratings;
+-----------+---------+------+-----+---------+-------+
| Field     | Type    | Null | Key | Default | Extra |
+-----------+---------+------+-----+---------+-------+
| userid    | int(11) | NO   | PRI | 0       |       |
| movieid   | int(11) | NO   | PRI | 0       |       |
| rating    | float   | YES  |     | NULL    |       |
| timestamp | int(11) | YES  |     | NULL    |       |
+-----------+---------+------+-----+---------+-------+
```

Figure 3.4: Ratings

4. **Tags**

```
mysql> SHOW COLUMNS FROM tags;
+-----------+-------------+------+-----+---------+----------------+
| Field     | Type        | Null | Key | Default | Extra          |
+-----------+-------------+------+-----+---------+----------------+
| userid    | int(11)     | YES  |     | NULL    |                |
| movieid   | int(11)     | YES  | MUL | NULL    |                |
| tag       | varchar(20) | YES  |     | NULL    |                |
| timestamp | int(11)     | YES  |     | NULL    |                |
| id        | int(11)     | NO   | PRI | NULL    | auto_increment |
+-----------+-------------+------+-----+---------+----------------+
```

Figure 3.5: Tags

At the same time, for processing the data in HBase it is required to merge tables before. As would be described later, we tried to use **Sqoop** to join our data on fly and put it to the table directly, but unfortunately, due to the lack of documentations, we could not fix it. So we decided to load the data to **MySql**, join it there and then move it to HBase. For better processing, we split the title and year from column *title* as well .

3.3 MySQL and Sqoop

First we did an experiment using Sqoop. We tried to join two tables at first and load it into HBase. Following is the command we have used:

> sqoop import –connect jdbc:mysql://localhost:3306/movielens –username root –password cloudera –query 'SELECT genres.id, genres.movieid, genres.genreid, movies.name, movies.year FROM genres, movies WHERE genres.movieid = movies.movieid AND $CONDITIONS' –hbase-create-table –hbase-table genres_movies –column-family c1 –split-by movie.id

But we met error when loading data:

Field	Type	Null	Key	Default	Extra
id	int(11)	NO		0	
userId	int(11)	YES		NULL	
movieId	int(11)	YES		NULL	
movieName	varchar(20)	YES		NULL	
movieYear	int(11)	YES		NULL	
movieTag	varchar(20)	YES		NULL	
tagTime	int(11)	YES		NULL	
userRating	float	YES		NULL	
ratingTime	int(11)	YES		NULL	
genresId	varchar(20)	YES		NULL	

Figure 3.6: Join results

Could not insert row with null value for row-key column: movie.id

Movie.id is the primary key of *Movies* table, there is no NULL value in the database. We did research on it and tried many solutions but still can not figure out the reason. Finally we gave up this method and join all the tables at MySql at first. Figure 3.6 is the schema of the final table we gained.

Chapter 4

Application

4.1 Project Set Up

For setting up a project we used an Eclipse IDE with Java 1.7 that is installed on the Cloudera CDH by default. For running the commands for HBase a number of different packages should be used. There are two different ways how we can include the packages. First one - using Java Build Path and add additional libraries. Unfortunately, this method does not work when the application should be run in the HDFS. For example, our team used this method at the beginning, but when we put it into the HDFS we started to face the problem with the dependencies. After careful search how to overcome the problem, we found a solution using Maven. Maven is a project management tool that allows to specify dependencies and run the project, Maven can find a library in the local repository or download it from a remote server if the library is not presented on the local computer. After adding Maven to our project we need to specify so called artefacts - dependencies that are required for the project. For instance, our second small project for Bulk Loading uses 7 different artefacts for handling Hadoop, HBase and tracking a logging. All the artefacts can be seen on Figure 4.1.

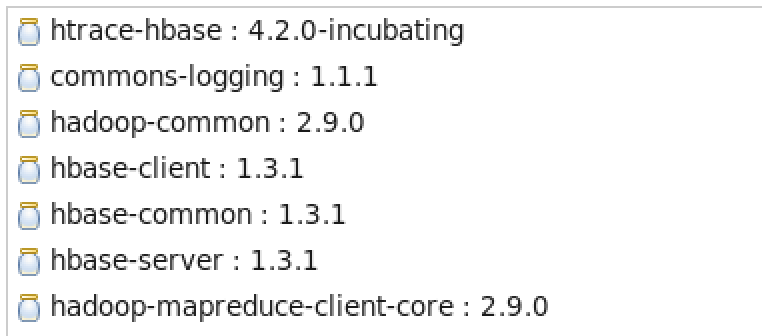

Figure 4.1: Project dependencies

After maven is added if project needs to be run in the HDFS than we need to export jar file that includes all the dependencies and put this file into HDFS and run it.

4.2 Basic projects and commands using Java API

In this section we would like to share some of the basic knowledge how to write operations to the HBase. We would like to cover a couple of CRUD operations as well as such examples as Database Creation, Delete and some other advanced techniques. We would like to start from the HBase table creation that meets us with basic imports and commands that are required for HBase administration. This project uses the same Maven dependencies as were shown in Section **Project Set Up**. On the Figure 4.2 we can see the minimum required listing to create the table and write a new column family into it.

```
public static void createTable() throws IOException{
    Configuration conf = HBaseConfiguration.create();
    Connection conn = ConnectionFactory.createConnection(conf);
    Admin admin = conn.getAdmin();
    HTableDescriptor table = new HTableDescriptor(TableName.valueOf("movies"));
    HColumnDescriptor family = new HColumnDescriptor("movieInfo");
    table.addFamily(family);
    admin.createTable(table);
}
```

Figure 4.2: Basic application that creates a table

We would like to describe some operators that are used in the code because they are used in almost every nowadays HBase Java project. First of all, *Configuration* and *Connection* are required for setting up a flexible connection. Type of the connection can be changed in the configuration file(for archiving this flexibility Factory design pattern is used [4]). Following that we create an instance

admin, that is required for storing a new table in the database. Following that, we create a table and a column family in memory, add the column to the table and call a method *createTable* from *admin*. The similar consequence without table creation is done if we want to delete the table. In this case the operator *deleteTable* is used. The method that deletes the table is presented on Figure 4.3.

```
public static void deleteTable(String tableName) throws IOException{
    Configuration conf = HBaseConfiguration.create();
    Connection conn = ConnectionFactory.createConnection(conf);
    Admin admin = conn.getAdmin();
    admin.deleteTables(tableName);
}
```

Figure 4.3: Method to delete the table

4.3 Data Loading

At first, we copied the data with movies in our Cloudera to the HDFS directory. We achieved it by the command listed on Figure 4.4.

```
[cloudera@quickstart ~]$ sudo hadoop fs -put /home/cloudera/Downloads/ml-20m/mov
ies data/ /user/cloudera/
```

Figure 4.4: Command to load the data into HDFS

Then the table movie is created. Table consists of 4 column families: *movieInfo*, *movieGenre*, *tag* and *rating*. We diveded column based on their data source(each column family is loaded from the different file) as well as distribution of values that would be assigned to one key. *movieInfo* consists of 2 fields *name* and *year*, this information never changes so number of **movieInfo Verstion** is set to **1**. At the same time, one movie can have many genres so **movieGenre Version** is set to 1. Of course, users can submit a thousand of *ratings* and *tags* so we set up the **Version** number of this column families to quite high value: **10000** and **500** accordingly. Please refer to Figure 4.5 for looking into the table details. For creating the table the method that is showed on Figure 4.6 was created. There we specify all the column families, add them to the table and create it.

```
{NAME => 'movieGenre', DATA_BLOCK_ENCODING => 'NONE', BLOOMFILTER => 'ROW', REPL
ICATION_SCOPE => '0', VERSIONS => '10', COMPRESSION => 'NONE', MIN_VERSIONS => '
0', TTL => 'FOREVER', KEEP_DELETED_CELLS => 'FALSE', BLOCKSIZE => '65536', IN_ME
MORY => 'false', BLOCKCACHE => 'true'}
{NAME => 'movieInfo', DATA_BLOCK_ENCODING => 'NONE', BLOOMFILTER => 'ROW', REPLI
CATION_SCOPE => '0', VERSIONS => '1', COMPRESSION => 'NONE', MIN_VERSIONS => '0'
, TTL => 'FOREVER', KEEP_DELETED_CELLS => 'FALSE', BLOCKSIZE => '65536', IN_MEMO
RY => 'false', BLOCKCACHE => 'true'}
{NAME => 'rating', DATA_BLOCK_ENCODING => 'NONE', BLOOMFILTER => 'ROW', REPLICAT
ION_SCOPE => '0', VERSIONS => '10000', COMPRESSION => 'NONE', MIN_VERSIONS => '0
', TTL => 'FOREVER', KEEP_DELETED_CELLS => 'FALSE', BLOCKSIZE => '65536', IN_MEM
ORY => 'false', BLOCKCACHE => 'true'}
{NAME => 'tag', DATA_BLOCK_ENCODING => 'NONE', BLOOMFILTER => 'ROW', REPLICATION
_SCOPE => '0', VERSIONS => '500', COMPRESSION => 'NONE', MIN_VERSIONS => '0', TT
L => 'FOREVER', KEEP_DELETED_CELLS => 'FALSE', BLOCKSIZE => '65536', IN_MEMORY =
> 'false', BLOCKCACHE => 'true'}
```

Figure 4.5: HBase table Column Families

```java
public static void createTable() throws IOException{
    Admin admin = getAdmin();
    HTableDescriptor movieTable = new HTableDescriptor(TableName.valueOf(tableName));
    HColumnDescriptor column1 = new HColumnDescriptor(columnFamily1).setMaxVersions(versic
    HColumnDescriptor column2 = new HColumnDescriptor(columnFamily2).setMaxVersions(versic
    HColumnDescriptor column3 = new HColumnDescriptor(columnFamily3).setMaxVersions(versic
    HColumnDescriptor column4 = new HColumnDescriptor(columnFamily4).setMaxVersions(versic
    movieTable.addFamily(column1);
    movieTable.addFamily(column2);
    movieTable.addFamily(column3);
    movieTable.addFamily(column4);
    admin.createTable(movieTable);
}
```

Figure 4.6: Method to create movies

For data loading the BulkLoad method was used with a configuration that is showed on Figures 4.7 and 4.8. Please notice that we specify a timestamp, it is done because during the bulk load data is written at the same time and versions mechanism in HBase is done on the base of different writing times. So in order to manage the system correctly, we need to set this parameter up explicitly.

```java
private static final String DATA_SEPERATOR = ",";
private static final String TABLE_NAME = "movies";
private static final String COLUMN_FAMILY_1="movieInfo";
private static final String COLUMN_FAMILY_2="movieGenre";
private static final String COLUMN_FAMILY_3="tag";
private static final String COLUMN_FAMILY_4="rating";
```

Figure 4.7: Parameters set up

```java
public void map(LongWritable key, Text value, Context context) {
    try {
        String[] values = value.toString().split(dataSeperator);
        String rowKey = values[1];
        Put put = new Put(Bytes.toBytes(rowKey));
        long timeStamp = Long.parseLong(values[3], 10);
        put.add(Bytes.toBytes(columnFamily4), Bytes.toBytes("userId"), timeStamp, Bytes.toBytes(values[0]));
        put.add(Bytes.toBytes(columnFamily4), Bytes.toBytes("rating"), timeStamp, Bytes.toBytes(values[2]));
        context.write(hbaseTableName, put);
    } catch(Exception exception) {
        exception.printStackTrace();
    }
}
```

Figure 4.8: HBase write configuration

After all the data was uploaded we made a comparison of time that each write batch operation took in order to finish it writing. Results can be seen on Figure 4.9. The time and size dependency should be linear but because of that *Rating* was written first and *Tag* contains many tags for the same movie, it is not.

Type	Size	Time
Rating	20.000.000	3537s
Tag	465565	643
Genres	54.407	61s
Movies	27229	25s

Figure 4.9: Insert operations compartion

If we need to select some data we can use Java API, for example, the following mapReduce method selects movie by movie name.

```java
public static void selectMovies(String tableName) throws IOException,ClassNotFoundException, InterruptedExceptic
    Configuration config = HBaseConfiguration.create();
    Job job = new Job(config, "ExampleRead");
    job.setJarByClass(SelectMoviesJob.class);     // class that contains mapper

    Scan scan = new Scan();
    scan.setCaching(10000);        // 1 is the default in Scan, which will be bad for MapReduce jobs
    scan.setCacheBlocks(false);
    scan.setAttribute("movieInfo:name", "Toy Story".getBytes());
    scan.setCacheBlocks(false);


    TableMapReduceUtil.initTableMapperJob(
      tableName,
      scan,
      SelectMoviesJob.MyMapper.class,
      null,
      null,
      job);
    job.setOutputFormatClass(NullOutputFormat.class);
```

Figure 4.10: Map Reduce Query

4.4 HBase benchmark comparison

For making a comparison between HBase and different databases we used a data from the paper **Comparison of database and workload types performance in Cloud environments** [7]. This paper compare the performance of basic operations: Read, Update Delete in HBase, Cassandra and MongoDb. The paper provide sufficient comparison of this databases in cloud. For example, it suggests that Hbase is the better option if database performs a big number of updates(Figure 4.11).

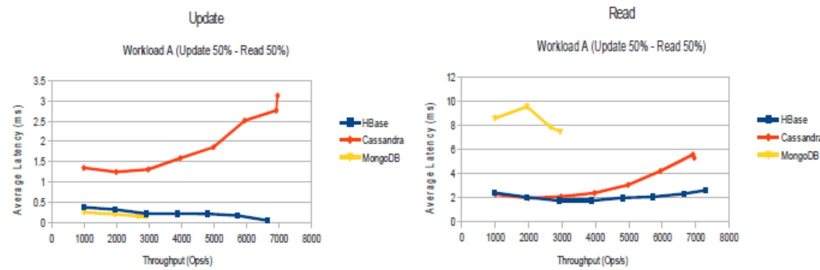

Figure 4.11: Databases write comparison

At the same time, if number of updates is small, performance is almost the same(Figure 4.12). Please use the paper for getting more precised information.

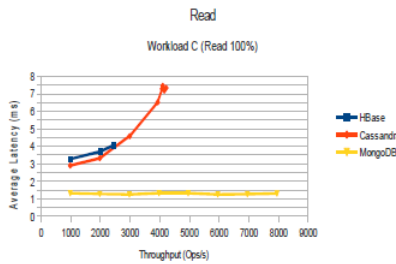

Figure 4.12: Database Read Comparison

Chapter 5

Conclusion

In this project we tried to learn new technologies that are widely used in the nowadays world. We learnt the following tools and technologies while prepare for the project:

1. Hadoop Core
2. HDFS
3. HBase
4. MapReduce
5. Maven
6. Sqoop
7. MySql

As well we measured a speed of huge data inserting into the database and created basics map-Reduce queries to get the data from the database. We also ran a couple of

Bibliography

[1] Movielens dataset. `https://grouplens.org/datasets/movielens/`.

[2] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, Fikes. A, and Gruber. R. *Bigtable: A Distributed Storage System for Structured Data*. Google, Inc, 2006.

[3] J. Dean and S. Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters*. Google, Inc, 2004.

[4] Erich Gamma. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[5] Lars George. *HBase: The Definitive Guide, Random Access to Your Planet-Size Data*. O'Reilly Media, 2001.

[6] S. Ghemawat, Gobioff H., and Shun-Tak Leung. *The Google File System*. Google, 2003.

[7] G. Seriatos, G. Kousiouris, A. Menychtas, Kyriazis D., and Varvarigou T. *Comparison of database and workload types performance in Cloud environments*. Sprinter, 2015.