A BigBench Implementation in the Hadoop Ecosystem

Badrul Chowdhury[1], Tilmann Rabl[1], Pooya Saadatpanah[2], Jiang Du[2], and Hans-Arno Jacobsen[1]

[1] Middleware Systems Research Group
[2] Database Research Group
University of Toronto
badrul.chowdhury@mail.utoronto.ca, tilmann.rabl@utoronto.ca, pooya@cs.toronto.edu, jdu@cs.toronto.edu, jacobsen@eecg.toronto.edu
http://msrg.org

Abstract. BigBench is the first proposal for an end to end big data analytics benchmark. It features a rich query set with complex, realistic queries. BigBench was developed based on the decision support benchmark TPC-DS. The first proof-of-concept implementation was built for the Teradata Aster parallel database system and the queries were formulated in the proprietary SQL-MR query language. To test other other systems, the queries have to be translated.
In this paper, an alternative implementation of BigBench for the Hadoop ecosystem is presented. All 30 queries of BigBench were realized using Apache Hive, Apache Hadoop, Apache Mahout, and NLTK. We will present the different design choices we took and show a proof of concept evaluation.

1 Introduction

Big data analytics is an ever growing field of research and business. Due to the drastic decrease of cost of storage and computation more and more data sources become profitable for data mining. A perfect example is online stores, while earlier online shopping systems would only record successful transactions, modern systems record every single interaction of a user with the website. The former allowed for simple basket analysis techniques, while current level of detail in monitoring makes detailed user modeling possible.

The growing demands on data management systems and the new forms of analysis have led to the development of a new breed of systems, big data management systems (BDMS) [1]. Similar to the advent of database management systems, there is a vastly growing ecosystem of diverse approaches. This leads to a dilemma for customers of BDMSs, since there are no realistic and proven measures to compare different offerings. To this end, we have developed BigBench, the first proposal for an end to end big data analytics benchmark [2]. BigBench was designed to cover essential functional and business aspects of big data use cases.

In this paper, we present an alternative implementation of the BigBench workload for the Hadoop eco-system. We re-implemented all 30 queries and ran proof of concept experiments on a 1 GB BigBench installation.

The rest of the paper is organized as follows. In Section 2, we present an overview of the BigBench benchmark. Section 3 introduces the parts of the Hadoop ecosystem that were used in our implementation. We give details on the transformation and implementation of the workload in Section 4. We present a proof of concept evaluation of our implementation in Section 5. Section 6 gives an overview of related work. We conclude with future work in Section 7.

2 BigBench Overview

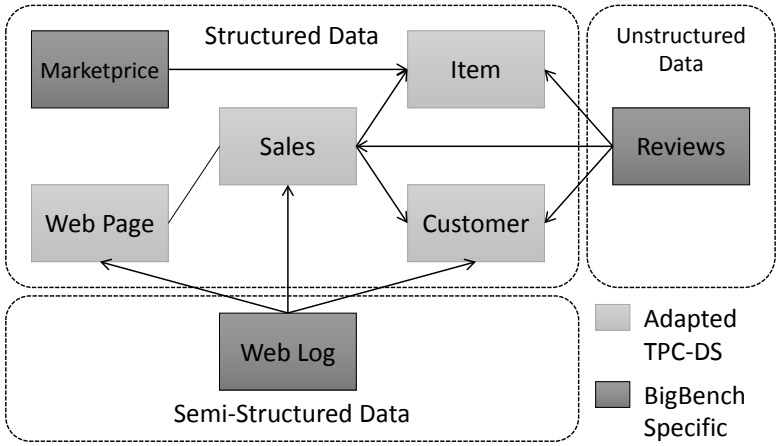

Fig. 1. BigBench Schema

BigBench is an end-to-end big data analytics benchmark, it was built to resemble modern analytic use cases in retail business. As basis for the benchmark, the Transaction Processing Performance Council's (TPC) new decision support benchmark TPC-DS was chosen [3]. This choice highly sped up the development of BigBench and made it possible to start from a solid and proven foundation. A high-level overview of the data model can be seen in Figure 1. The TPC-DS data model is a snowflake schema with 6 fact tables, representing 3 sales channels, store sales, catalog sales, and online sales, each with a sales and a returns fact table. For BigBench the catalog sales were removed, since they have decreasing significance in retail business. As can be seen in Figure 1, additional big data specific dimensions were added. Marketprice is a traditional relational table storing competitors prices. The Web Log portion represents a click-stream that is used to analyze the user behavior. This part of the data set is semi-structured, since different entries in the weblog represent different user actions

and thus have different format. The log is generated in form of an Apache Web server log. The unstructured part of the schema is generated in form of product reviews. These are, for example, used for sentiment analysis. The full schema is described in [4].

BigBench features 30 complex queries, 10 of which are taken from TPC-DS, the others were specifically developed for BigBench. The queries are covering major areas of big data analytics as specified in [5]. As a result, the queries cannot be expressed by pure SQL queries since they include machine learning techniques, sentiment analysis, and procedural computations. In Teradata Aster, this is solved using built in functions that are internally processed in a MapReduce fashion. The benchmark, however, does not dictate a specific implementation, therefore, the benchmark can be implemented in various ways. The full list of queries can be found in [4].

3 Technologies for BigBench on Hadoop

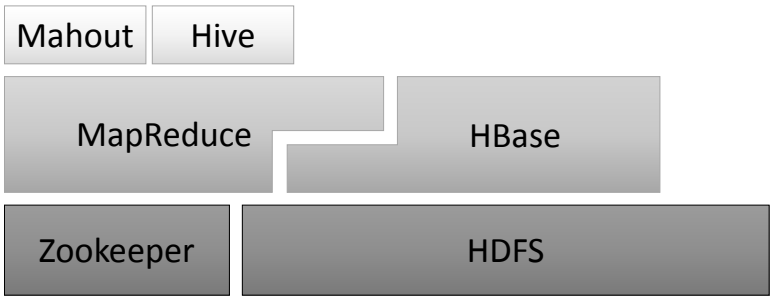

Fig. 2. Hadoop Stack

In this section, the technologies used to create an open-source implementation of BigBench are described. BigBench is mainly implemented using four open-source software frameworks: Apache Hadoop, Apache Hive, Apache Mahout, and the Natural Language Processing Toolkit (NLTK). We used the following versions for our implementation: Apache Hadoop V0.20.2, Apache Hive 0.8.1, Apache Mahout 0.6, and NLTK 3.0.

3.1 Hadoop

Apache Hadoop provides a scalable distributed file system and features to perform analysis on and store large data sets using the MapReduce framework [6]. Its architecture consists of many components and a discussion of the design decisions with implementation details can be found [7]. Only components that are most relevant to the BigBench implementation will be described in the following.

The *Hadoop Distributed File System* (HDFS) is modeled after the Unix file system hierarchy with 3-way replication of data for security and analysis performance purposes. The Hadoop command line interface provides access to a most standard Unix file operations such as *ls, rm, cp*, etc. A complete reference can be found on Apache Hadoop's website[3].

A cluster implementing HDFS has 3 main components: *HDFS client, namenode,* and *datanode*. The namenode primarily stores meta data. It keeps a record of the *namespace tree*, which stores information relevant to file block allocation to datanode. It should be noted that all of the namespace data is stored in RAM. There can only be one namenode in any single cluster in the version of Hadoop used. However, on the other hand, there are usually multiple datanode in a cluster. Each datanode contains two files in the local file system: one to store metadata and the other to store the actual data. The HDFS client provides an interface for user-created applications to access and modify HDFS. Access is provided in a two-tiered process: first, the metadata in the namenode is extracted and then information is used to access the relevant datanodes.

3.2 Hive

Apache Hive is a data warehousing solution being developed with a view to provide traditional SQL programmers access to the functionality of MapReduce [8]. To this end, Hive features an SQL-like structured language called Hive Query Language (HiveQL)[4]. The version used in the BigBench implementation supports a proper subset of standard SQL operations in addition to providing the facility to "stream" custom user-defined MapReduce jobs to perform operations on stored tables.

We will only outline the basic architecture here, a more detailed description can be found in [8]. Some components of the Hive architecture that are worth of noticing are the *metastore*, the *query compiler*, and the *execution engine*. The metastore is used to store all operation-critical information pertaining to table schema, table location, table columns and their data types. The information stored in the metastore needs to be extracted very quickly for data analysis and transformation and, therefore, it is usually stored in a local database. The most commonly used metastore system is MySQL. The query compiler compiles the queries written in Hive-QL and optimizes the queries where possible. Finally, the execution engine that is based on MapReduce executes the tasks specified by the compiled queries according to their precedence in the dependency tree.

3.3 Mahout and NLTK

Apache Mahout is an open-source community effort to build a scalable machine learning algorithm library on top of Apache Hadoop[5]. It has an ever-increasing

[3] `http://hadoop.apache.org/`
[4] http://hive.apache.org/
[5] `https://mahout.apache.org/`

number of machine learning classification and clustering algorithms among many others. Mahout is designed to run such algorithms on distributed file systems. In the BigBench implementation, Mahout is used mainly to run the *k-means* algorithm on HDFS.

NLTK provides a library of Python functions for the processing of natural language using standard statistical techniques [9]. It is distributed under the Apache License[6]. NLTK is used to implement sentiment analysis in BigBench queries.

4 Query Implementation

In this section, the implementation of the different flavors of queries will be discussed. The overall number of 30 queries has been grouped into 4 categories: Pure Hive queries, Hive queries with MapReduce programs, Hive queries using natural language processing, and queries using Apache Mahout. In the following, we will give an example for each of the different flavors of queries.

The distribution of the different query types is shown in Table 1. Table 2 shows the data types that the queries access as specified in Section 2.

Data Type	**BigBench Queries**	**Percentage**
Pure Hive	5, 6, 7, 9, 11, 12, 13, 14, 16, 17, 21, 22, 23, 24	46.7
Hive + MR	1, 2, 3, 4, 8	16.7
Hive + Hadoop Streaming	15, 18	6.7
Mahout	20, 25, 26, 29, 30	16.7
NLTK	10, 19, 27, 28	13.2

Table 1. Distribution of BigBench Queries by Query Type

Data Type	**BigBench Queries**	**Percentage**
Structured	1, 6, 7, 9, 13, 14, 15, 16, 17, 19, 20, 21, 22, 23, 24, 25, 26, 29	60.0
Semi-Structured	2, 3, 4, 5, 8, 12, 30	23.3
Unstructured	10, 11, 18, 27, 28	16.7

Table 2. Distribution of BigBench Queries by Data Type

It should be noted that queries that use NLTK and Mahout also require preprocessing by Hive. Therefore, Apache Hive is critical to all data processing activities in this implementation of BigBench.

[6] http://www.nltk.org/

4.1 Loading Data

The synthetically generated data is loaded onto Hive in 2 steps:

1. Each table required by the benchmark is created in Hive using the syntax shown in Listing 1.1. The full list of table descriptions is presented in [4].
2. The data is loaded onto the Hive tables using the directive in Listing 1.2.

```
drop table if exists customer_demographics;

create table customer_demographics
 ( cd_demo_sk                bigint,
   cd_gender                 string,
   cd_marital_status         string,
   cd_education_status       string,
   cd_purchase_estimate      int,
   cd_credit_rating          string,
   cd_dep_count              int,
   cd_dep_employed_count     int,
   cd_dep_college_count      int
 )
ROW FORMAT DELIMITED FIELDS TERMINATED BY '|';
```

Listing 1.1. Creating the *customer_demographics* table in Hive

```
LOAD DATA LOCAL INPATH 'customer_demographics.dat'
     OVERWRITE INTO TABLE customer_demographics;
```

Listing 1.2. Loading *customer_demographics* data onto Hive

Data is loaded in the same way for all of the 24 tables of the benchmark.

4.2 Hive Queries

Hive does not support the full standard SQL syntax; some of these peculiarities will be discussed in more detail in this section as relevant to BigBench Query 21.

Query 21 (TPC-DS 29) :

> Get all items that were sold in stores in a given month and year and which were returned in the next six months and re-purchased by the returning customer afterwards through the web sales channel in the following three years. For those these items, compute the total quantity sold through the store, the quantity returned and the quantity purchased through the web. Group this information by item and store.

```
SELECT i_item_id, i_item_desc, s_store_id, s_store_name,
       sum(ss_quantity) AS store_sales_quantity,
       sum(sr_return_quantity) AS store_returns_quantity,
       sum(ws_quantity) AS web_sales_quantity
  FROM store_sales, store_returns, web_sales, date_dim d1,
       date_dim d2, date_dim d3, store, item
 WHERE d1.d_moy             = 4
   AND d1.d_year            = 1998
   AND d1.d_date_sk         = ss_sold_date_sk
   AND i_item_sk            = ss_item_sk
   AND s_store_sk           = ss_store_sk
   AND ss_customer_sk       = sr_customer_sk
   AND ss_item_sk           = sr_item_sk
   AND ss_ticket_number     = sr_ticket_number
   AND sr_returned_date_sk  = d2.d_date_sk
   AND d2.d_moy             BETWEEN 4 AND  4 + 3
   AND d2.d_year            = 1998
   AND sr_customer_sk       = ws_bill_customer_sk
   AND sr_item_sk           = ws_item_sk
   AND ws_sold_date_sk      = d3.d_date_sk
   AND d3.d_year            IN (1998,1998+1,1998+2)
 GROUP BY i_item_id, i_item_desc, s_store_id, s_store_name
 ORDER BY i_item_id, i_item_desc, s_store_id, s_store_name;
```

Listing 1.3. Query 21 - Aster SQL-MR Syntax

```
SELECT * FROM
  SELECT i.i_item_id AS item_id, i.i_item_desc AS item_desc,
         s.s_store_id AS store_id, s.s_store_name AS store_name,
         SUM(ss.ss_quantity) AS store_sales_quantity,
         SUM(sr.sr_return_quantity) AS store_returns_quantity,
         SUM(ws.ws_quantity) AS web_sales_quantity
    FROM store_sales ss
         JOIN item i ON (i.i_item_sk = ss.ss_item_sk)
         JOIN store s ON (s.s_store_sk = ss.ss_store_sk)
         JOIN date_dim d1 ON (d1.d_date_sk = ss.ss_sold_date_sk
                AND d1.d_moy = 4 AND d1.d_year = 1998)
         JOIN store_returns s ON (ss.ss_customer_sk = sr.sr_customer_sk
                AND ss.ss_item_sk = s.sr_item_sk)
         JOIN date_dim d2 ON (sr.sr_returned_date_sk = d2.d_date_sk
                AND d2.d_moy > 4-1 AND d2.d_moy < 4+3+1 AND d2.d_year = 1998)
         JOIN web_sales ws ON (sr.sr_item_sk = ws.ws_item_sk)
         JOIN date_dim d3 ON (ws.ws_sold_date_sk = d3.d_date_sk
                AND d3.d_year IN (1998 ,1998+1 ,1998+2))
 GROUP BY i.i_item_id, i.i_item_desc,  s.s_store_id, s.s_store_name)
 ORDER BY item_id, item_desc, store_id, store_name;
```

Listing 1.4. Query 21 - Hive Syntax

Query 21 is taken from TPC-DS and thus a traditional relational query. The version in SQL-MR syntax is shown in Listing 1.3. It joins 8 tables, 3 fact tables and 5 dimensions. The largest table, store_sales is joined with item, store, and date_dim to find items bought in a particular month. Then, by joining with store_returns, the items that were returned are filtered. Finally, using a join with web_sales items that were previously returned in store and then bought again online within 3 years are selected. These are grouped by by item and store. The Hive version, as shown in Listing 1.4, is an almost word-for-word translation of the SQL-MR version with a few notable differences in syntax. The Hive implementation makes use of the `JOIN` syntax extensively; this is due to the fact that the current Hive version only supports a single table in the `FROM`-clause, which may be a composite of multiple tables itself, as in this case. Also, arguments of the `WHERE`-clause in the SQL-MR query have been used as

JOIN conditions in the Hive version on grounds of improving efficiency: the Hive version's implementation eliminates the need to loop over the resulting table again after the joins have taken place.

4.3 Hive and MapReduce

Data processing tasks becomes simpler and more intuitive using custom programs in several BigBench queries. External programs are therefore used in the queries by means of Hive's program streaming feature to mimic some of SQL-MR's built-in functions. This is discussed with on the example of Query 10 of BigBench. Query 10 is an example of sentiment analysis, which is not included in the standard SQL functionality.

Query 10 :

> For all products, extract sentences from its product reviews that contain positive or negative sentiment and display the sentiment polarity of the extracted sentences.

```
SELECT pr_item_sk, out_content, out_polarity, out_sentiment_words
  FROM ExtractSentiment
       (ON product_reviews100
        TEXT_COLUMN ('pr_review_content')
        MODEL ('dictionary')
        LEVEL ('sentence')
        ACCUMULATE ('pr_item_sk')
       )
 WHERE out_polarity = 'NEG'
    OR out_polarity = 'POS';
```

Listing 1.5. Query 10 - SQL-MR Syntax

```
ADD FILE mapper_10.py;
ADD FILE reducer_10.py;

FROM (
  FROM   product_reviews
    MAP    product_reviews.pr_item,
                  product_reviews.pr_review_content
    USING 'python␣mapper_10.py'
    AS item, polarity
) mapper

REDUCE mapper.item, mapper.polarity
  USING 'python␣reducer_10.py'
  AS (item STRING, polarity STRING);
```

Listing 1.6. Query 10 - Hive Syntax

In the mapper and reducer files are imported using the ADD FILE filename directive. The mapper is invoked using the USING LANGUAGE MAPPER_FILE directive; the reducer is used similarly. The mapper should generate a (KEY, VALUE) vector as per map-reduce guidelines: this vector is generated using the MAP KEY, VALUE directive. Later, the REDUCE KEY, VALUE directive is used to reduce the mapped vector.

4.4 Hive and Natural Language Processing

All of the natural language processing capabilities of the Hive implementation of BigBench are implemented using the Natural Language Toolkit (NLTK) 3.0 package. In this section, the NLP-processing capability of Query 10 is analyzed in more detail.

All of the features of the NLTK package are available in the program after it is imported using the standard Python `import` syntax.

```
import nltk
[..]
def get_word_features(wordlist):
  wordlist = nltk.FreqDist(wordlist)
  word_features = wordlist.keys()
  return word_features

def extract_sentiment(tweet):
  [..]
  for (words, sentiment) in pos_tweets + neg_tweets:
      words_filtered = [e.lower() for e in words.split() if len(e) >= 3]
      tweets.append((words_filtered, sentiment))
  training_set = nltk.classify.apply_features(extract_features, tweets)
  classifier = nltk.NaiveBayesClassifier.train(training_set)
  return classifier.classify(extract_features(tweet.split()))
[..]
```

Listing 1.7. Query 10 - Sentiment_Analysis.py Program

The package provides powerful APIs for natural language processing. An excellent example of such an API is `nltk.FreqDist(DOCUMENT)`, as used in the sentiment analysis in Listing 1.7. The `FreqDist` method gets a Python set as parameter and returns the modified set containing the frequency distribution of each word in the original input set. The training and application of the NLTK classifier on a body of text is done in several steps. First, the `nltk.classify.apply_features()` method is used to apply a "negative" or "positive" label to each feature of the training data (the `extract_features` method extracts the features from the list of tweets). Then, the classifier is trained using Naive Bayes by invoking the method `nltk.NaiveBayesClassifier train(.training_set)`. Finally, the trained classifier is used to label the features of any new body of text by invoking `classifier.classify(extract_features(tweet.split()))`.

The overall quality of the results depends on the size and quality of the training data. In the current version of the implementation, the model is trained on a very small hand-made data set; a future improvement to the model will train it on larger training sets to improve its repertoire of feature labels.

Programs to analyze natural language have thereby been used (using streaming in Hive) to add features such as sentiment analysis to some Hive queries in this implementation.

4.5 Mahout

Apache Mahout is used to implement all of the machine-learning capabilities of BigBench; this will be exemplified with reference to Query 20 of BigBench in

this section. The SQL-MR implementation of Query 20 can be seen in Listing 1.8.

Query 20 :

> Customer segmentation for return analysis: Customers are separated along the following dimensions: return frequency, return order ratio (total number of orders partially or fully returned versus the total number of orders), return item ratio (total number of items returned versus the number of items purchased), return amount ration (total monetary amount of items returned versus the amount purchased), return order ratio. Consider the store returns during a given year for the computation.

```
CREATE VIEW sales_returns AS (
  SELECT s.ss_sold_date_sk AS s_date,
         r.sr_returned_date_sk AS r_date,
         s.ss_item_sk AS item,
         s.ss_ticket_number AS oid,
         s.ss_net_paid AS s_amount,
         r.sr_return_amt AS r_amount,
         (CASE WHEN s.ss_customer_sk IS NULL
               THEN r.sr_customer_sk ELSE s.ss_customer_sk END) AS cid,
         s.ss_customer_sk AS s_cid,
         sr_customer_sk AS r_cid
    FROM store_sales s LEFT JOIN store_returns100 r ON
        s.ss_item_sk = r.sr_item_sk
         AND s.ss_ticket_number = r.sr_ticket_number
   WHERE s.ss_sold_date_sk IS NOT NULL);

CREATE VIEW clusters AS (
  SELECT cid,
         100.0 * COUNT (DISTINCT (CASE WHEN r_date IS NOT NULL
                                       THEN oid ELSE NULL END))
               / COUNT (DISTINCT oid) AS r_order_ratio,
         SUM (CASE WHEN r_date IS NOT NULL THEN 1 ELSE 0 END)
               / COUNT (item) * 100 AS r_item_ratio,
         SUM (CASE WHEN r_date IS NOT NULL THEN r_amount ELSE 0 END)
               / SUM (s_amount) * 100 AS r_amount_ratio,
         COUNT (DISTINCT (CASE WHEN r_date IS NOT NULL
                               THEN r_date ELSE NULL END))
                               AS r_freq
    FROM sales_returns
   WHERE cid IS NOT NULL
   GROUP BY 1
  HAVING COUNT (DISTINCT (CASE WHEN r_date IS NOT NULL
                               THEN r_date ELSE NULL END)) > 1);

SELECT *
  FROM kmeans (ON
       (SELECT 1)
       PARTITION BY 1
       DATABASE ('benchmark')
       USERID ('benchmark')
       PASSWORD ('benchmark')
       INPUTTABLE ('clusters␣AS␣c')
       OUTPUTTABLE ('user_return_groups')
       NUMBERK('4'));

SELECT clusterid, cid
  FROM kmeansplot (ON
       clusters AS c
       PARTITION BY ANY
       ON user_return_groups dimension
```

```
      CENTROIDSTABLE ('user_return_groups'))
 ORDER BY clusterid, cid;

DROP TABLE user_return_groups;
DROP VIEW clusters;
DROP VIEW sales_returns;
```

Listing 1.8. Query 20 - SQL-MR Syntax

```
CREATE VIEW IF NOT EXISTS sales_returns AS
  SELECT s.ss_sold_date_sk AS s_date,
    r.sr_returned_date_sk AS r_date,
    s.ss_item_sk AS item,
    s.ss_ticket_number AS oid,
    s.ss_net_paid AS s_amount,
    r.sr_return_amt AS r_amount,
    (CASE WHEN s.ss_customer_sk IS NULL
          THEN r.sr_customer_sk ELSE s.ss_customer_sk END) AS cid,
    s.ss_customer_sk AS s_cid,
    sr_customer_sk AS r_cid
  FROM store_sales s
  LEFT OUTER JOIN store_returns r ON s.ss_item_sk = r.sr_item_sk AND
      s.ss_ticket_number = r.sr_ticket_number
  WHERE s.ss_sold_date_sk IS NOT NULL;

CREATE TABLE IF NOT EXISTS all_sales_returns AS
  SELECT * FROM sales_returns;

CREATE VIEW IF NOT EXISTS clusters AS
  SELECT cid,
    100.0 * COUNT(distinct(CASE WHEN r_date IS NOT NULL
                                THEN oid ELSE NULL end))
          / COUNT(distinct oid) AS r_order_ratio,
    SUM(CASE WHEN r_date IS NOT NULL
             THEN 1 ELSE 0 END)
          / COUNT(item)*100 AS r_item_ratio,
    SUM(CASE WHEN r_date IS NOT NULL
             THEN r_amount ELSE 0.0 END)
          / SUM(s_amount)*100 AS r_amount_ratio,
    COUNT(distinct (CASE WHEN r_date IS NOT NULL
                         THEN r_date ELSE NULL END)) AS r_freq
  FROM all_sales_returns
  WHERE cid IS NOT NULL
GROUP BY cid;

DROP TABLE IF EXISTS twenty;

CREATE TABLE IF NOT EXISTS twenty
  ROW FORMAT DELIMITED
  FIELDS TERMINATED BY '␣'
  LINES TERMINATED BY '\n'
  STORED AS TEXTFILE LOCATION '/mahout_io/twenty/' AS
  SELECT * FROM clusters;

DROP TABLE IF EXISTS all_sales_returns;
```

Listing 1.9. Query 20 - Part 1 in Hive Syntax

The Hive/Mahout implementation of Query 20 can be seen in Listings 1.9 and 1.10. First the table `clusters` is created using Hive. It contains the fields on which segmentation analysis will be performed, namely `return_order_ratio, return_item_ratio, return_amount_ratio, return_frequency`. It is stored as a single whitespace character-delimited text file which is used as the input file to the Mahout k-means program that is shown in Listing 1.10

```
public class Twenty {
  public static void writePointsToFile( List<Vector> points, String fileName,
      FileSystem fs, Configuration conf ) throws IOException {
    Path path = new Path(fileName);
    SequenceFile.Writer writer = new SequenceFile.Writer(fs, conf, path,
      LongWritable.class, VectorWritable.class);
    long recNum = 0;
    VectorWritable vec = new VectorWritable();
    for ( Vector point : points ) {
      vec.set(point);
      writer.append(new LongWritable(recNum++), vec);
    }
    writer.close();
  }

  public static List<Vector> getPoints( double[][] tuples ) {
    List<Vector> points = new ArrayList<Vector>();
    for ( int i = 0; i < tuples.length; i++ ) {
      double[] fr = tuples[i];
      Vector vec = new RandomAccessSparseVector(fr.length);
      vec.assign(fr);
      points.add(vec);
    }
    return points;
  }

  public static void main( String args[] ) throws IOException {
    [..]
    // number of centres is 4 as per Query 20
    int k = 4;
    List<Vector> vectors = getPoints(myPoints);
    File tuples = new File("tuples");
    if ( !tuples.exists() ) {
      tuples.mkdir();
    }
    tuples = new File("tuples/points");
    if ( !tuples.exists() ) {
      tuples.mkdir();
    }
    Configuration conf = new Configuration();
    FileSystem fs = FileSystem.get(conf);
    writePointsToFile(vectors,"tuples/points/file1", fs, conf);
    Path path = new Path("tuples/clusters/part-00000");
    SequenceFile.Writer writer = new SequenceFile.Writer(fs, conf,
      path, Text.class, Cluster.class);
    for ( int i = 0; i < k; i++ ) {
      Vector vec = vectors.get(i);
      Cluster cluster = new Cluster(vec, i, new EuclideanDistanceMeasure());
      writer.append(new Text(cluster.getIdentifier()), cluster);
    }
    writer.close();
    KMeansDriver.run(conf, new Path("tuples/points"), new
      Path("tuples/clusters"), new Path("output"), new
      EuclideanDistanceMeasure(), 0.001, 10, true, false);
    SequenceFile.Reader reader = new SequenceFile.Reader(fs,
      new Path("output/" + Cluster.CLUSTERED_POINTS_DIR + "/part-m-00000"),
      conf);
    IntWritable key = new IntWritable();
    WeightedPropertyVectorWritable value = new
        WeightedPropertyVectorWritable();
    [..]
    reader.close();
  }
}
```

Listing 1.10. Query 20 - Part 2 as a Mahout Program

The k-means clustering algorithm in Mahout has a very specific control flow. First, each input tuple (which is a single line in the input *clusters* file) is converted into a vector; the cardinality of the tuple is preserved by this operation. In the Java program, the `getPoints()` method does this by creating a `RandomAccessSparseVector` for each tuple. Then, these vectors are written to a file in the specified input directory; the `writePointsToFile()` method does this creating `VectorWritable` and `SequenceFileWriter` objects to create the writable representation of the vector and to perform the write-operation respectively. In this process, a Hadoop-specific data type –`LongWritable`– is used. Finally, the points are clustered in several passes over the input vectors and the output after each pass is stored in separate subdirectories within the `output` directory. In the program, this final step is commenced by running the static function `run()` from the `KMeansDriver` class, which takes in the similarity measure to be used to perform the clustering (in this case, the `EuclideanDistanceMeasure`) as a parameter.

The program is called from the command line using Hadoop streaming; the program is run as shown in Listing 1.11. However, before using the Hadoop streaming feature, the respective Mahout libraries must be added to the `HADOOP_CLASSPATH` environment variable. The exact libraries to be included for this particular program are shown in listing 1.11.

```
$ export
    HADOOP_CLASSPATH=~/mahout/mahout-core-0.6.jar:~/mahout/mahout-math-0.
    6.jar:~/mahout/mahout-core-0.6-job.jar
$ /home/usr/hadoop-0.20.2/bin/hadoop jar Twenty.jar Twenty
    /mahout_io/Twenty/000000_0 /mahout_io/Twenty
```

Listing 1.11. Running K-means Program of Query 20 Using Hadoop Streaming

5 Evaluation

We run a proof-of-concept evaluation on a single setup. The system was fitted with 5.8 GB of RAM, a 750 GB SATA hard-disk, and a 3.2 GHz Intel Xeon Quadcore processor. Each query was run on a 1.3 GB data set, the results are shown in Table 5. The *System Runtime* attribute corresponds to the time obtained by using the Unix *time* command when running the queries. The *Reported Time* attribute corresponds to the time reported by Hive and Mahout respectively; it should be noted that for queries which require running multiple engines, the numbers in the table correspond to the sum of the partial running time of each engine.

The results are plotted in Figure 3. In general, the Unix *time* utility reported a higher time than the internal time reporting feature of Hive and Mahout. The internal time reporting feature of both Hive and Mahout displays the time it takes to complete a specified Map-Reduce job, so the difference between the two depicted times corresponds to the *set-up* and *tear-down* time of the system for each query.

In particular, Query 13 takes a noticeably longer time than the other queries due to the presence of many `JOIN` statements. Each `JOIN` statement is translated

Query	System Runtime/s	Reported Time/s	Query	Actual Runtime/s	Reported Time/s
1	97	96	2	58	56
3	38	37	4	279	275
5	34	33	6	294	264
7	298	282	8	51	47
9	185	174	10	13	11
11	69	60	12	105	174
13	4694	4373	14	219	216
15	844	783	16	170	109
17	349	347	18	642	548
19	91	88	20	206	157
21	273	271	22	293	292
23	634	498	24	195	193
25	202	160	26	62	60
27	55	52	28	32	30
29	294	288	30	301	296

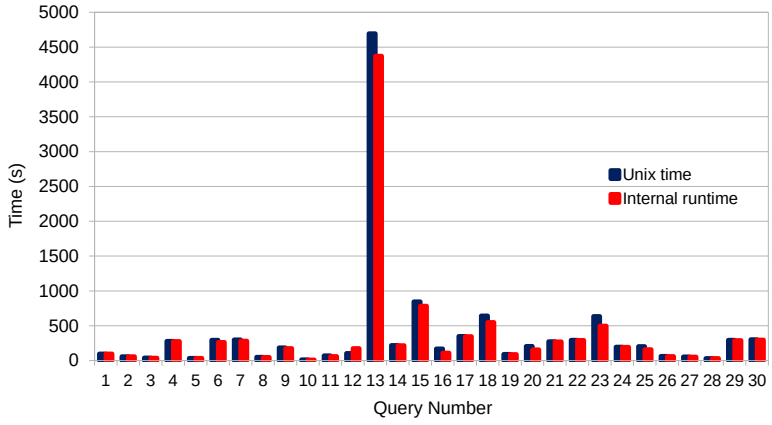

Fig. 3. Query Runtime

to a map-reduce job by Hive based on the join condition, and since the query was run on a single node, the query execution engine incurred considerable loss in time due to the *set-up* and *tear-down* time consumed by each individual job.Also, the tables that are joined in this query are much richer in content than in the other queries.

6 Related Work

TPC benchmarks are commonly used for benchmarking big data systems. For big data analytics, TPC-H and TPC-DS are obvious choices and TPC-H has been implemented in Hadoop, Pig[7], and Hive[8] [10, 11]. A subset of TPC-DS has recently been used to compare DBMSes with Impala and Hive[9]. However, TPC-H and TPC-DS are pure SQL benchmarks and thus do not cover all the different aspects that MapReduce systems are typically used for. Several proposals try to modify TPC-DS similar to BigBench to cover typical big data use cases. Zhao et al. propose Big DS, which extends the TPC-DS model with social marketing and advertisement [12]. Currently, Big DS is in a very early design stage and no query set and data model are available. Once the benchmark has matured, it should be possible to complement BigBench with the Big DS proposal. Another TPC-DS modification is proposed by Yi and Dai as part of the HiBench suite [13, 14]. The authors use the TPC-DS model to generate web logs similar to BigBench. Unlike BigBench the authors use this for an ETL process. This again is orthogonal to BigBench and can be included in future work. There have been several other proposals, most of which are component benchmarks testing specific functions of the big data systems. Two notable examples are the Berkeley Big Data Benchmark[10] and the benchmark presented by Pavlo et al. [15]. Another example is BigDataBench, which is a suite similar to HiBench and mainly targeted at hardware benchmarking [16]. Although interesting and very useful, these benchmarks do not present an end to end scenario and thus have another focus than BigBench.

7 Conclusion

BigBench is the only fully specified end to end benchmark for big data analytics currently available. In this paper, we presented details about our ongoing implementation for the Hadoop ecosystem. The implementation is completely based on open-source libraries and frameworks typically used in big data deployments. The queries and the data set can be downloaded from the MSRG website[11].

For future work, we will work on improving the data generation and the data model. We are currently building a complete kit for measuring the end to end

[7] `https://issues.apache.org/jira/browse/PIG-2397`
[8] `https://issues.apache.org/jira/browse/HIVE-600`
[9] `http://blog.cloudera.com/blog/2014/01/impala-performance-dbms-class-speed/`
[10] `https://amplab.cs.berkeley.edu/benchmark/`
[11] `http://msrg.org`

processing time including loading and refresh. We will investigate the inclusion of other proposals such as the ETL-pipeline proposed as part of the HiBench suite [14].

References

1. Carey, M.J.: BDMS Performance Evaluation: Practices, Pitfalls, and Possibilities. In Nambiar, R., Poess, M., eds.: Selected Topics in Performance Evaluation and Benchmarking. Volume 7755 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2013) 108–123
2. Ghazal, A., Rabl, T., Hu, M., Raab, F., Poess, M., Crolotte, A., Jacobsen., H.A.: BigBench: Towards an industry standard benchmark for big data analytics. In: Proceedings of the ACM SIGMOD Conference. (2013)
3. Pöss, M., Nambiar, R.O., Walrath, D.: Why You Should Run TPC-DS: A Workload Analysis. In: VLDB. (2007) 1138–1149
4. Rabl, T., Ghazal, A., Hu, M., Crolotte, A., Raab, F., Poess, M., Jacobsen, H.A.: BigBench Specification V0.1. In Rabl, T., Poess, M., Baru, C., Jacobsen, H.A., eds.: Specifying Big Data Benchmarks. Volume 8163 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2014) 164–201
5. Manyika, J., Chui, M., Brown, B., Bughin, J., Dobbs, R., Roxburgh, C., Byers, A.H.: Big data: The Next Frontier for Innovation, Competition, and Productivity. Technical report, McKinsey Global Institute (2011) `http://www.mckinsey.com/insights/mgi/research/technology_and_innovation/big_data_the_next_frontier_for_innovation`.
6. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. Communications of the ACM **51**(1) (2008) 107–113
7. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The Hadoop Distributed File System. In: 26th IEEE Symposium on Mass Storage Systems and Technologies. (2010) 1–10
8. Thusoo, A., Sarma, J.S., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., Murthy, R.: Hive: A Warehousing Solution Over a Map-Reduce Framework. Proceedings of the VLDB Endowment **2**(2) (2009) 1626–1629
9. Bird, S., Klein, E., Loper, E., Baldridge, J.: Multidisciplinary Instruction with the Natural Language Toolkit. In: Proceedings of the Third Workshop on Issues in Teaching Computational Linguistics. TeachCL '08 (2008) 62–70
10. Moussa, R.: TPC-H Benchmark Analytics Scenarios and Performances on Hadoop Data Clouds. In Benlamri, R., ed.: Networked Digital Technologies. Volume 293 of Communications in Computer and Information Science. Springer Berlin Heidelberg (2012) 220–234
11. Kim, K., Jeon, K., Han, H., gyu Kim, S., Jung, H., Yeom, H.: MRBench: A Benchmark for MapReduce Framework. In: Parallel and Distributed Systems, 2008. ICPADS '08. 14th IEEE International Conference on. (Dec 2008) 11–18
12. Zhao, J.M., Wang, W., Liu, X.: Big Data Benchmark - Big DS. In: Proceedings of the Third and Fourth Workshop on Big Data Benchmarking 2013. (2014) (in print).
13. Huang, S., Huang, J., Dai, J., Xie, T., Huang, B.: The HiBench Benchmark Suite: Characterization of the MapReduce-Based Data Analysis. In: ICDEW. (2010)
14. Yi, L., Dai, J.: Experience from Hadoop Benchmarking with HiBench: from Micro-Benchmarks toward End-to-End Pipelines. In: Proceedings of the Third and Fourth Workshop on Big Data Benchmarking 2013. (2014) (in print).

15. Pavlo, A., Paulson, E., Rasin, A., Abadi, D.J., DeWitt, D.J., Madden, S., Stonebraker, M.: A Comparison of Approaches to Large-Scale Data Analysis. In: SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data. (2009) 165–178
16. Wang, L., Zhan, J., Luo, C., Zhu, Y., Yang, Q., He, Y., Gao, W., Jia, Z., Shi, Y., Zhang, S., Zhen, C., Lu, G., Zhan, K., Li, X., Qiu, B.: BigDataBench: a Big Data Benchmark Suite from Internet Services. In: Proceedings of the 20th IEEE International Symposium On High Performance Computer Architecture. HPCA (2014)